

# Python Language advantages and applications

Python is a high-level, interpreted, and general-purpose dynamic programming language that focuses on code readability. It has fewer steps when compared to Java and C. It was founded in 1991 by developer Guido Van Rossum. Python ranks among the most popular and fastest-growing languages in the world. Python is a powerful, flexible, and easy-to-use language. In addition, the community is very active there. It is used in many organizations as it supports multiple programming paradigms. It also performs automatic memory management.

## Advantages :

1. Presence of third-party modules
2. Extensive support libraries(NumPy for numerical calculations, Pandas for data analytics etc)
3. Open source and community development
4. Versatile, Easy to read, learn and write
5. User-friendly data structures
6. High-level language
7. Dynamically typed language(No need to mention data type based on the value assigned, it takes data type)
8. Object-oriented language
9. Portable and Interactive
10. Ideal for prototypes – provide more functionality with less coding
11. Highly Efficient(Python's clean object-oriented design provides enhanced process control, and the language is equipped with excellent text processing and integration capabilities, as well as its own unit testing framework, which makes it more efficient.)
12. (IoT)Internet of Things Opportunities
13. Interpreted Language
14. Portable across Operating systems

## Applications :

1. GUI based desktop applications
2. Graphic design, image processing applications, Games, and Scientific/ computational Applications
3. Web frameworks and applications
4. Enterprise and Business applications
5. Operating Systems
6. Education
7. Database Access
8. Language Development
9. Prototyping
10. Software Development

## Organizations using Python :

1. Google(Components of Google spider and Search Engine)
2. Yahoo(Maps)
3. YouTube
4. Mozilla
5. Dropbox
6. Microsoft
7. Cisco
8. Spotify
9. Quora

'import' keyword is used to import a particular module into your python code.

We can also get all the keyword names using the below code.

[Example: Python Keywords List](#)

```
# Python code to demonstrate working of iskeyword()
```

```
# importing "keyword" for keyword operations
```

```
import keyword
```

```
# printing all keywords at once using "kwlist()"
```

```
print("The list of keywords is : ")
```

```
print(keyword.kwlist)
```

**Output:**

The list of keywords is :

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

```
# using is to check object identity
```

```
# string is immutable( cannot be changed once allocated)
```

```
# hence occupy same memory location
```

```
print(' ' is ' ')
```

```
# using is to check object identity
```

```
# dictionary is mutable( can be changed once allocated)
```

```
# hence occupy different memory location
```

```
print({} is {})
```

```
True
```

```
False
```

## Return Keywords – Return, Yield

- [return](#) : This keyword is used to return from the function.
- [yield](#) : This keyword is used like return statement but is used to return a generator.

[Example: Return and Yield Keyword](#)

```
# Return keyword
```

```
def fun() :
```

```
    S = 0
```

```
    for i in range(10) :
```

```
        S += i
```

```
    return S
```

```
print(fun())

# Yield Keyword

def fun():
    s = 0
    for i in range(10):
        s += i
        yield s
for i in fun():
    print(i)
```

#### Output

```
45
0
1
3
6
10
15
21
28
36
45
```

## Python return statement

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned.

**Note:** Return statement can not be used outside the function.

#### Syntax:

```
def fun():
    statements
    return [expression]
```

#### Example:

```
# Python program to
# demonstrate return statement
```

```
def add(a, b):

    # returning sum of a and b
    return a + b

def is_true(a):

    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))

res = is_true(2<5)
print("\nResult of is_true function is {}".format(res))
```

### Output:

```
Result of add function is 5
Result of is_true function is True
```

## Returning Multiple Values

In Python, we can return multiple values from a function. Following are different ways.

- **Using Object:** This is similar to C/C++ and Java, we can create a class (in C, struct) to hold multiple values and return an object of the class.

```
# A Python program to return multiple
# values from a method using class
class Test:

    def __init__(self):

        self.str = "geeksforgeeks"

        self.x = 20
```

```
# This function returns an object of Test
```

```
def fun():
```

```
    return Test()
```

```
# Driver code to test above method
```

```
t = fun()
```

```
print(t.str)
```

```
print(t.x)
```

- **Output:**

```
geeksforgeeks
```

```
20
```

- **Using Tuple:** A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable. See [this](#) for details of [tuple](#).

```
# A Python program to return multiple
```

```
# values from a method using tuple
```

```
# This function returns a tuple
```

```
def fun():
```

```
    str = "geeksforgeeks"
```

```
    x = 20
```

```
    return str, x; # Return tuple, we could also
```

```
                # write (str, x)
```

```
# Driver code to test above method
```

```
str, x = fun() # Assign returned tuple
```

```
print(str)
```

```
print(x)
```

- **Output:**

```
geeksforgeeks
```

```
20
```

- **Using a list:** A list is like an array of items created using square brackets. They are different from arrays as they can contain items of different types. Lists are different from tuples as they are mutable.

```
# A Python program to return multiple
# values from a method using list
```

```
# This function returns a list
```

```
def fun():
    str = "geeksforgeeks"
    x = 20
    return [str, x];
```

```
# Driver code to test above method
```

```
list = fun()
print(list)
```

- Output:

```
['geeksforgeeks', 20]
```

- Using a Dictionary: A Dictionary is similar to hash or map in other languages.

```
# A Python program to return multiple
# values from a method using dictionary
```

```
# This function returns a dictionary
```

```
def fun():
    d = dict();
    d['str'] = "GeeksforGeeks"
    d['x'] = 20
    return d
```

```
# Driver code to test above method
```

```
d = fun()
print(d)
```

- Output:

```
{'x': 20, 'str': 'GeeksforGeeks'}
```

## Function returning another function

In Python, functions are objects so, we can return a function from another function. This is possible because functions are treated as first class objects in Python. In the below example, the `create_adder` function returns `adder` function.

```
# Python program to illustrate functions
# can return another function
```

```
def create_adder(x):
    def adder(y):
        return x + y

    return adder
```

```
add_15 = create_adder(15)
```

```
print("The result is", add_15(10))
```

```
# Returning different function
```

```
def outer(x):
    return x * 10
```

```
def my_func():
```

```
    # returning different function
    return outer
```

```
# storing the function in res
```

```
res = my_func()
```

```
print("\nThe result is:", res(10))
```

### Output:

```
The result is 25
The result is: 100
```

# Python | yield Keyword

**Yield** is a keyword in Python that is used to return from a function without destroying the states of its local variable and when the function is called, the execution starts from the last yield statement. Any function that contains a yield keyword is termed a generator. Hence, yield is what makes a generator. The yield keyword in Python is less known off but has a greater utility which one can think of.

**Code #1 :** Demonstrating yield working

```
# Python3 code to demonstrate
# yield keyword

# generator to print even numbers
def print_even(test_list) :
    for i in test_list:
        if i % 2 == 0:
            yield i

# initializing list
test_list = [1, 4, 5, 6, 7]

# printing initial list
print("The original list is : " + str(test_list))

# printing even numbers
print("The even numbers in list are : ", end = " ")
for j in print_even(test_list):
    print(j, end = " ")
```

**Output :**

```
The original list is : [1, 4, 5, 6, 7]
The even numbers in list are :  4 6
```

**Code #2:**



```

# A Python program to generate squares from 1
# to 100 using yield and therefore generator

# An infinite generator function that prints
# next square number. It starts with 1
def nextSquare():
    i = 1

    # An Infinite loop to generate squares
    while True:
        yield i*i
        i += 1 # Next execution resumes
              # from this point

# Driver code
for num in nextSquare():
    if num > 100:
        break
    print(num)

```

### Output:

```

1
4
9
16
25
36
49
64
81
100

```

### Advantages of yield:

- Since it stores the local variable states, hence overhead of memory allocation is controlled.
- Since the old state is retained, the flow doesn't start from the beginning and hence saves time.

### Disadvantages of yield:

- Sometimes, the use of yield becomes erroneous if the calling of function is not handled properly.

- Time and memory optimization has a cost of complexity of code and hence sometimes hard to understand the logic behind it.

### **Practical Applications:**

The possible practical application is that when handling the last amount of data and searching particulars from it, yield can be used as we don't need to look up again from start and hence would save time. There can possibly be many applications of yield depending upon the use cases.

```
# Python3 code to demonstrate yield keyword

# Checking number of occurrence of
# geeks in string

# generator to print even numbers
def print_even(test_string) :
    for i in test_string:
        if i == "geeks":
            yield i

# initializing string
test_string = " The are many geeks around you, \
              geeks are known for teaching other geeks"

# printing even numbers using yield
count = 0
print ("The number of geeks in string is : ", end = "" )
test_string = test_string.split()

for j in print_even(test_string):
    count = count + 1

print (count)
```

### **Output :**

```
The number of geeks in string is : 3
```

# with statement in Python

`with` statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Observe the following code example on how the use of `with` statement makes code cleaner.

```
# file handling

# 1) without using with statement
file = open('file_path', 'w')
file.write('hello world !')
file.close()

# 2) without using with statement
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()

# using with statement
with open('file_path', 'w') as file:
    file.write('hello world !')
```

Notice that unlike the first two implementations, there is no need to call `file.close()` when using `with` statement. The `with` statement itself ensures proper acquisition and release of resources. An exception during the `file.write()` call in the first implementation can prevent the file from closing properly which may introduce several bugs in the code, i.e. many changes in files do not go into effect until the file is properly closed.

The second approach in the above example takes care of all the exceptions but using the `with` statement makes the code compact and much more readable. Thus, `with` statement helps avoiding bugs and leaks by ensuring that a resource is properly released when the code using the resource is completely executed. The `with` statement is popularly used with file streams, as shown above and with Locks, sockets, subprocesses and telnet etc.

## Supporting the “with” statement in user defined objects

There is nothing special in `open()` which makes it usable with the `with` statement and the same functionality can be provided in user defined objects. Supporting `with` statement in your objects will ensure that you never leave any resource open.

To use `with` statement in user defined objects you only need to add the methods `__enter__()` and `__exit__()` in the object methods. Consider the following example for further clarification.

```
# a simple file writer object

class MessageWriter(object):

    def __init__(self, file_name):
        self.file_name = file_name

    def __enter__(self):
        self.file = open(self.file_name, 'w')
        return self.file

    def __exit__(self):
        self.file.close()

# using with statement with MessageWriter

with MessageWriter('my_file.txt') as xfile:

    xfile.write('hello world')
```

Let's examine the above code. If you notice, what follows the `with` keyword is the constructor of `MessageWriter`. As soon as the execution enters the context of the `with` statement a `MessageWriter` object is created and python then calls the `__enter__()` method. In this `__enter__()` method, initialize the resource you wish to use in the object. This `__enter__()` method should always return a descriptor of the acquired resource.

### What are resource descriptors?

These are the handles provided by the operating system to access the requested resources. In the following code block, `file` is a descriptor of the file stream resource.

```
file = open('hello.txt')
```

In the `MessageWriter` example provided above, the `__enter__()` method creates a file descriptor and returns it. The name `xfile` here is used to refer to the file descriptor returned

by the `__enter__()` method. The block of code which uses the acquired resource is placed inside the block of the `with` statement. As soon as the code inside the `with` block is executed, the `__exit__()` method is called. All the acquired resources are released in the `__exit__()` method. This is how we use the `with` statement with user defined objects.

This interface of `__enter__()` and `__exit__()` methods which provides the support of `with` statement in user defined objects is called ***Context Manager***.

#### The `contextlib` module

A class based context manager as shown above is not the only way to support the `with` statement in user defined objects. The `contextlib` module provides a few more abstractions built upon the basic context manager interface. Here is how we can rewrite the context manager for the `MessageWriter` object using the `contextlib` module.

```
from contextlib import contextmanager

class MessageWriter(object):

    def __init__(self, filename):
        self.file_name = filename

    @contextmanager
    def open_file(self):
        try:
            file = open(self.file_name, 'w')
            yield file
        finally:
            file.close()

# usage
message_writer = MessageWriter('hello.txt')
with message_writer.open_file() as my_file:
    my_file.write('hello world')
```

In this code example, because of the `yield` statement in its definition, the function `open_file()` is a [generator function](#).

When this `open_file()` function is called, it creates a resource descriptor named `file`. This resource descriptor is then passed to the caller and is represented here by the variable `my_file`. After the code inside the `with` block is executed the program control returns back to the `open_file()` function. The `open_file()` function resumes its execution and executes

the code following the `yield` statement. This part of code which appears after the `yield` statement releases the acquired resources. The `@contextmanager` here is a [decorator](#).

The previous class-based implementation and this generator-based implementation of context managers is internally the same. While the later seems more readable, it requires the knowledge of generators, decorators and `yield`.

## as

**as** keyword is used to create the alias for the module imported. i.e giving a new name to the imported module. E.g `import math as mymath`.

Example: **as** Keyword

```
import math as gfg
```

```
print(gfg.factorial(5))
```

Output

120

## Import, From

- **import** : This statement is used to include a particular module into current program.
- **from** : Generally used with `import`, `from` is used to import particular functionality from the module imported.

Example: **Import, From** Keyword

```
# import keyword
```

```
import math
```

```
print(math.factorial(10))
```

```
# from keyword
```

```
from math import factorial
```

```
print(factorial(10))
```

Output

3628800

3628800

# finally keyword in Python

Prerequisites: [Exception Handling](#), [try and except in Python](#)

In programming, there may be some situation in which the current method ends up while handling some exceptions. But the method may require some additional steps before its termination, like closing a file or a network and so on.

So, in order to handle these situations, Python provides a keyword **finally**, which is always executed after [try and except](#) blocks. The **finally** block always executes after normal termination of *try block* or after *try block* terminates due to some exception.

## Syntax:

```
try:
    # Some Code....

except:
    # optional block
    # Handling of exception (if required)

finally:
    # Some code .....(always executed)
```

## Important Points –

- *finally* block is always executed after leaving the *try* statement. In case if some exception was not handled by *except* block, it is re-raised after execution of *finally* block.
- *finally* block is used to deallocate the system resources.
- One can use *finally* just after *try* without using *except* block, but no exception is handled in that case.

## Example #1:

```
# Python program to demonstrate finally

# No exception Exception raised in try block
try:
    k = 5//0 # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # This block is always executed regardless of exception or not
```

```
# this block is always executed
# regardless of exception generation.
print('This is always executed')
```

### Output:

```
Can't divide by zero
This is always executed
```

### Example #2:

```
# Python program to demonstrate finally

try:
    k = 5//1 # No exception raised
    print(k)

# intends to handle zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

### Output:

```
5
This is always executed
```

### Example #3:

```
# Python program to demonstrate finally

# Exception is not handled
try:
    k = 5//0 # exception raised
```



```

print(k)

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')

```

### Output:

This is always executed

### Runtime Error –

```

Unhandled Exception
k=5//0 #No exception raised
ZeroDivisionError: integer division or modulo by zero

```

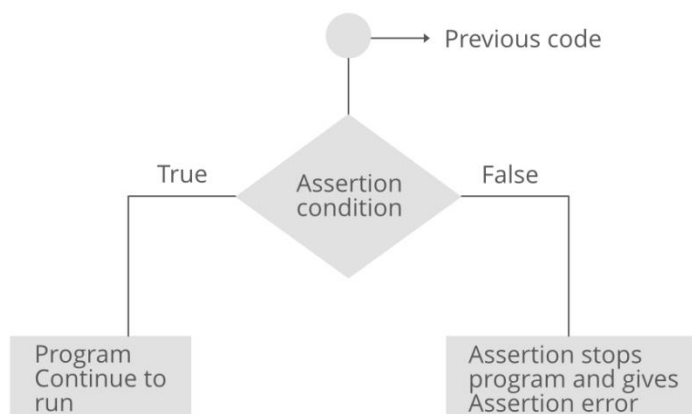
### Explanation:

In above code, the exception is generated *integer division or modulo by zero*, which was not handled. The exception was re-raised after execution of *finally* block. This shows that *finally* block is executed regardless of exception is handled or not.

## Python assert keyword

**Assertions** in any programming language are the debugging tools that help in the smooth flow of code. Assertions are mainly assumptions that a programmer knows always wants to be true and hence puts them in code so that failure of them doesn't allow the code to execute further.

In simpler terms, we can say that assertion is the boolean expression that checks if the statement is True or False. If the statement is true then it does nothing and continues the execution but if the statement is False then it stops the execution of the program and throws an error. Let us look at the flowchart of the assertion.



## Flowchart of Assertion

### Assert Keyword in Python

In python assert keyword helps in achieving this task. This statement simply takes input a boolean condition, which when returns true doesn't return anything, but if it is computed to be false, then it raises an AssertionError along with the optional message provided.

**Syntax :** assert condition, error\_message(optional)

**Parameters :**

**condition :** The boolean condition returning true or false.

**error\_message :** The optional argument to be printed in console in case of AssertionError

**Returns :**

Returns AssertionError, in case the condition evaluates to false along with the error message which when provided.

**Example 1: Python assert keyword without error message**

**# Python 3 code to demonstrate**

**# working of assert**

**# initializing number**

**a = 4**

**b = 0**

**# using assert to check for 0**

**print("The value of a / b is : ")**

**assert b != 0**

**print(a / b)**

**Output :**

**AssertionError:**

**Example 2: Python assert keyword with error message**

**# Python 3 code to demonstrate**

**# working of assert**

**# initializing number**

```
a = 4
b = 0

# using assert to check for 0
print("The value of a / b is : ")
assert b != 0, "Zero Division Error"
print(a / b)
```

### Output:

AssertionError: Zero Division Error

### Practical Application

This has a much greater utility in testing and Quality assurance role in any development domain. Different types of assertions are used depending upon the application. Below is the simpler demonstration of a program that only allows only the batch with all hot food to be dispatched, else rejects the whole batch.

```
# Python 3 code to demonstrate
# working of assert
# Application

# initializing list of foods temperatures
batch = [ 40, 26, 39, 30, 25, 21]

# initializing cut temperature
cut = 26

# using assert to check for temperature greater than cut
for i in batch:
    assert i >= 26, "Batch is Rejected"
    print (str(i) + " is O.K" )
```

### Output :

```
40 is O.K
26 is O.K
39 is O.K
30 is O.K
```

## Runtime Exception :

`AssertionError: Batch is Rejected`

# Global keyword in Python

Global keyword is a keyword that allows a user to modify a variable outside of the current scope. It is used to create [global variables](#) from a non-global scope i.e inside a function. Global keyword is used inside a function only when we want to do assignments or when we want to change a variable. Global is not needed for printing and accessing.

## Rules of global keyword:

- If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.
- Variables that are only referenced inside a function are implicitly global.
- We Use global keyword to use a global variable inside a function.
- There is no need to use global keyword outside a function.

## Use of global keyword:

To access a global variable inside a function there is no need to use global keyword.

### Example 1:

```
# Python program showing no need to
# use global keyword for accessing
# a global value

# global variable
a = 15
b = 10

# function to perform addition
def add():
    c = a + b
    print(c)

# calling a function
add()
```

## Output:

25

If we need to assign a new value to a global variable then we can do that by declaring the variable as global.

**Code 2:** Without global keyword

```
# Python program showing to modify
# a global value without using global
# keyword

a = 15

# function to change a global value
def change():

    # increment value of a by 5
    a = a + 5
    print(a)

change()
```

**Output:**

UnboundLocalError: local variable 'a' referenced before assignment

This output is an error because we are trying to assign a value to a variable in an outer scope. This can be done with the use of **global** variable.

**Code 3 :** With global keyword

```
# Python program to modify a global
# value inside a function

x = 15

def change():

    # using a global keyword
    global x

    # increment value of a by 5
    x = x + 5
```

```

    print("Value of x inside a function :", x)

change()

print("Value of x outside a function :", x)

```

### Output:

```

Value of x inside a function : 20
Value of x outside a function : 20

```

**In the above example, we first define x as global keyword inside the function `change()`.** The value of x is then incremented by 5, ie.  $x=x+5$  and hence we get the output as 20. As we can see by changing the value inside the function `change()`, the change is also reflected in the value outside the global variable.

### Global variables across python modules :

The best way to share global variables across different modules within the same program is to create a special module (often named config or cfg). Import the config module in all modules of your application; the module then becomes available as a global name. There is only one instance of each module and so any changes made to the module object get reflected everywhere. For Example, sharing global variables across modules

**Code 1:** Create a `config.py` file to store global variables:

```

x = 0
y = 0
z = "none"

```

**Code 2:** Create a `modify.py` file to modify global variables:

```

import config

config.x = 1
config.y = 2
config.z = "geeksforgeeks"

```

Here we have modified the value of x, y, and z. These variables were defined in the module `config.py`, hence we have to import `config` module and we can use `config.variable_name` to access these variables.

**Code 3:** Create a `main.py` file to modify global variables:

```

import config
import modify

print(config.x)
print(config.y)
print(config.z)

```

## Output:

```
1
2
geeksforgeeks
```

## Global in Nested functions

In order to use global inside a nested functions, we have to declare a variable with global keyword inside a nested function

```
# Python program showing a use of
# global in nested function

def add() :
    x = 15

    def change() :
        global x
        x = 20

        print("Before making changing: ", x)
        print("Making change")
        change()
        print("After making change: ", x)

add()

print("value of x",x)
```

## Output:

```
Before making changing:  15
Making change
After making change:  15
value of x 20
```

In the above example Before and after making change(), the variable `x` takes the value of local variable i.e `x = 15`. Outside of the `add()` function, the variable `x` will take value defined in the `change()` function i.e `x = 20`. because we have used global keyword in `x` to create global variable inside the `change()` function (local scope).

## Global, Nonlocal

- **global:** This keyword is used to define a variable inside the function to be of a global scope.
- **non-local :** This keyword works similar to the global, but rather than global, this keyword declares a variable to point to variable of outside enclosing function, in case of nested functions.

Example: Global and nonlocal keywords

```
# global variable

a = 15

b = 10


# function to perform addition
def add():
    c = a + b
    print(c)


# calling a function
add()


# nonlocal keyword
def fun():
    var1 = 10

    def gun():
        # tell python explicitly that it
        # has to access var1 initialized
        # in fun on line 2
        # using the keyword nonlocal
        nonlocal var1

        var1 = var1 + 10
        print(var1)

    gun()

fun()
```

Output

```
25
20
```

### *Statements*

Instructions written in the source code for execution are called statements. There are different types of statements in the Python programming language like Assignment statements,



Conditional statements, Looping statements, etc. These all help the user to get the required output. For example, `n = 50` is an assignment statement.

**Multi-Line Statements:** Statements in Python can be extended to one or more lines using parentheses `()`, braces `{}`, square brackets `[]`, semi-colon `(;)`, continuation character slash `(\)`. When the programmer needs to do long calculations and cannot fit his statements into one line, one can make use of these characters.

**Example :**

Declared using Continuation Character `(\)` :

```
s = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

Declared using parentheses `()` :

```
n = (1 * 2 * 3 + 7 + 8 + 9)
```

Declared using square brackets `[]` :

```
footballer = ['MESSI',
              'NEYMAR',
              'SUAREZ']
```

Declared using braces `{}` :

```
x = {1 + 2 + 3 + 4 + 5 + 6 +
     7 + 8 + 9}
```

Declared using semicolons `(;)` :

```
flag = 2; ropes = 3; pole = 4
```

## Taking multiple inputs from user in Python

The developer often wants a user to enter multiple values or inputs in one line. In C++/C user can take multiple inputs in one line using `scanf` but in Python user can take multiple values or inputs in one line by two methods.

- Using `split()` method
- Using List comprehension

Using [`split\(\)`](#) method :

## Python String | `split()`

`split()` method in Python split a string into a list of strings after breaking the given string by the specified separator.

**Syntax :** `str.split(separator, maxsplit)`

**Parameters :**

**separator :** This is a delimiter. The string splits at this specified separator. If is not provided then any white space is a separator.

**maxsplit :** It is a number, which tells us to split the string into maximum of provided number of times. If it is not provided then the default is -1 that means there is no limit.

**Returns :** Returns a list of strings after breaking the given string by the specified separator.

**Example 1:** Example to demonstrate how split() function works

```
text = 'geeks for geeks'

# Splits at space
print(text.split())

word = 'geeks, for, geeks'

# Splits at ','
print(word.split(','))

word = 'geeks:for:geeks'

# Splitting at ':'
print(word.split(':'))

word = 'CatBatSatFatOr'

# Splitting at t
print(word.split('t'))
```

**Output :**

```
['geeks', 'for', 'geeks']
['geeks', ' for', ' geeks']
['geeks', 'for', 'geeks']
['Ca', 'Ba', 'Sa', 'Fa', 'Or']
```

**Example 2:** Example to demonstrate how split() function works when maxsplit is specified

```
word = 'geeks, for, geeks, pawan'

# maxsplit: 0
```

```
print(word.split(' ', 0))
```

```
# maxsplit: 4
```

```
print(word.split(' ', 4))
```

```
# maxsplit: 1
```

```
print(word.split(' ', 1))
```

### Output :

```
['geeks', 'for', 'geeks', 'pawan']  
['geeks', 'for', 'geeks', 'pawan']  
['geeks', 'for', 'geeks', 'pawan']
```

This function helps in getting multiple inputs from users. It breaks the given input by the specified separator. If a separator is not provided then any white space is a separator. Generally, users use a split() method to split a Python string but one can use it in taking multiple inputs.

### Syntax :

```
input().split(separator, maxsplit)
```

### Example :

```
# Python program showing how to  
# multiple input using split
```

```
# taking two inputs at a time  
x, y = input("Enter two values: ").split()  
print("Number of boys: ", x)  
print("Number of girls: ", y)  
print()
```

```
# taking three inputs at a time  
x, y, z = input("Enter three values: ").split()  
print("Total number of students: ", x)  
print("Number of boys is : ", y)  
print("Number of girls is : ", z)
```

```

print()

# taking two inputs at a time
a, b = input("Enter two values: ").split()
print("First number is {} and second number is {}".format(a, b))
print()

# taking multiple inputs at a time
# and type casting using list() function
x = list(map(int, input("Enter multiple values: ").split()))
print("List of students: ", x)

```

### Output:

```

Enter a two value: 5 10
Number of boys: 5
Number of girls: 10

Enter a three value: 30 10 20
Total number of students: 30
Number of boys is : 10
Number of girls is : 20

Enter a four value: 20 30
First number is 20 and second number is 30

Enter a multiple value: 20 30 10 22 23 26
List of students: [20, 30, 10, 22, 23, 26]
...

```

### Using [List comprehension](#) :

List comprehension is an elegant way to define and create list in Python. We can create lists just like mathematical statements in one line only. It is also used in getting multiple inputs from a user.

### Example:

```

# Python program showing
# how to take multiple input
# using List comprehension

# taking two input at a time
x, y = [int(x) for x in input("Enter two values: ").split()]

```

```

print("First Number is: ", x)
print("Second Number is: ", y)
print()

# taking three input at a time
x, y, z = [int(x) for x in input("Enter three values: ").split()]
print("First Number is: ", x)
print("Second Number is: ", y)
print("Third Number is: ", z)
print()

# taking two inputs at a time
x, y = [int(x) for x in input("Enter two values: ").split()]
print("First number is {} and second number is {}".format(x, y))
print()

# taking multiple inputs at a time
x = [int(x) for x in input("Enter multiple values: ").split()]
print("Number of list is: ", x)

```

## Output :

```

Enter two value: 2 5
First Number is: 2
Second Number is: 5

Enter three value: 2 4 5
First Number is: 2
Second Number is: 4
Third Number is: 5

Enter two value: 2 10
First number is 2 and second number is 10

Enter multiple value: 1 2 3 4 5
Number of list is: [1, 2, 3, 4, 5]
~~~

```

**Note:** The above examples take input separated by spaces. In case we wish to take input separated by comma (, ), we can use the following:

```

# taking multiple inputs at a time separated by comma

```

```
x = [int(x) for x in input("Enter multiple value: ").split(",")]  
print("Number of list is: ", x)
```

## Python | Get a list as input from user

We often encounter a situation when we need to take number/string as input from the user. In this article, we will see how to get as input a list from the user.

### Examples:

```
Input : n = 4,  ele = 1 2 3 4  
Output : [1, 2, 3, 4]
```

```
Input : n = 6,  ele = 3 4 1 7 9 6  
Output : [3, 4, 1, 7, 9, 6]
```

### Code #1: Basic example

```
# creating an empty list  
lst = []  
  
# number of elements as input  
n = int(input("Enter number of elements : "))  
  
# iterating till the range  
for i in range(0, n):  
    ele = int(input())  
  
    lst.append(ele) # adding the element  
  
print(lst)
```

### Output:

```
Enter number of elements : 5  
12  
33  
0  
5  
13  
[12, 33, 0, 5, 13]
```

### Code #2: With handling exception

```
# try block to handle the exception
try:
    my_list = []

    while True:
        my_list.append(int(input()))

# if the input is not-integer, just print the list
except:
    print(my_list)
```

**Output:**

```
Enter the elements:
5
3
15
33
0
7
56
Stop
[5, 3, 15, 33, 0, 7, 56]
```

**Code #3:** Using map()

```
# number of elements
n = int(input("Enter number of elements : "))

# Below line read inputs from user using map() function
a = list(map(int,input("\nEnter the numbers : ").strip().split()))[:n]

print("\nList is - ", a)
```

**Output:**

```
Enter number of elements : 5

Enter the numbers : 3 15 22 18 33

List is -  [3, 15, 22, 18, 33]
```

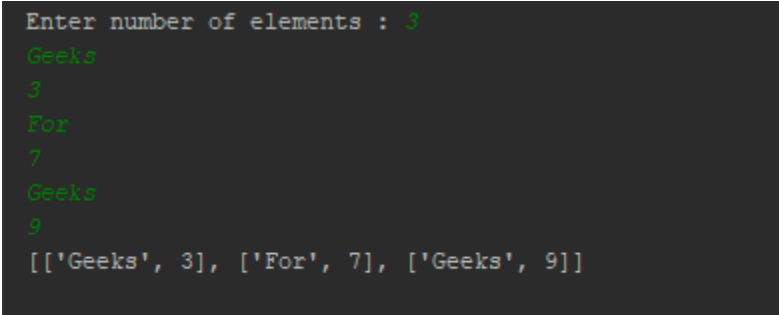
**Code #4:** List of lists as input

```
lst = [ ]
n = int(input("Enter number of elements : "))
```

```
for i in range(0, n):
    ele = [input(), int(input())]
    lst.append(ele)

print(lst)
```

**Output:**



```
Enter number of elements : 3
Geeks
3
For
7
Geeks
9
[['Geeks', 3], ['For', 7], ['Geeks', 9]]
```

**Code #5:** Using List Comprehension and Typecasting

```
# For list of integers
lst1 = []

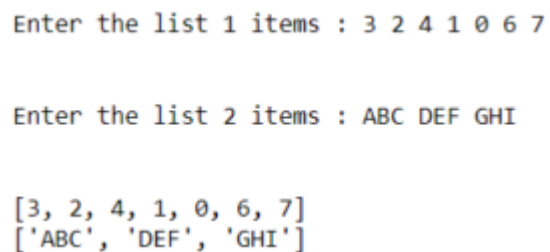
# For list of strings/chars
lst2 = []

lst1 = [int(item) for item in input("Enter the list items : ").split()]

lst2 = [item for item in input("Enter the list items : ").split()]

print(lst1)
print(lst2)
```

**Output:**



```
Enter the list 1 items : 3 2 4 1 0 6 7

Enter the list 2 items : ABC DEF GHI

[3, 2, 4, 1, 0, 6, 7]
['ABC', 'DEF', 'GHI']
```



# Python List Comprehension and Slicing

List comprehension is an elegant way to define and create a list in python. We can create lists just like mathematical statements and in one line only. The syntax of list comprehension is easier to grasp.

A list comprehension generally consists of these parts :

1. Output expression,
2. Input sequence,
3. A variable representing a member of the input sequence and
4. An optional predicate part.

For example :

```
lst = [x ** 2 for x in range (1, 11) if x % 2 == 1]
```

here, `x ** 2` is output expression,  
`range (1, 11)` is input sequence,  
`x` is variable and  
`if x % 2 == 1` is predicate part.

## Example 1:

```
# Python program to demonstrate list comprehension in Python
```

```
# below list contains square of all odd numbers from
```

```
# range 1 to 10
```

```
odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]
```

```
print (odd_square)
```

```
# for understanding, above generation is same as,
```

```
odd_square = []
```

```
for x in range(1, 11):
```

```
    if x % 2 == 1:
```

```
        odd_square.append(x**2)
```

```
print (odd_square)
```

```
# below list contains power of 2 from 1 to 8
```

```
power_of_2 = [2 ** x for x in range(1, 9)]
```

```
print (power_of_2)
```

```

# below list contains prime and non-prime in range 1 to 50
noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
primes = [x for x in range(2, 50) if x not in noprimes]
print (primes)

# list for lowering the characters
print ([x.lower() for x in ["A","B","C"]] )

# list which extracts number
string = "my phone number is : 11122 !!"

print("\nExtracted digits")
numbers = [x for x in string if x.isdigit()]
print (numbers)

# A list of list for multiplication table
a = 5
table = [[a, b, a * b] for b in range(1, 11)]

print("\nMultiplication Table")
for i in table:
    print (i)

```

## Output:

```

[1, 9, 25, 49, 81]
[1, 9, 25, 49, 81]
[2, 4, 8, 16, 32, 64, 128, 256]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
['a', 'b', 'c']

```

```

Extracted digits
['1', '1', '1', '2', '2']

```

```

Multiplication Table
[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]

```

```
[5, 9, 45]
[5, 10, 50]
```

After getting the list, we can get a part of it using python's slicing operator which has the following syntax:

```
[start : stop : steps]
```

which means that slicing will start from index **start** will go up to **stop** in **step** of steps.  
Default value of start is 0, stop is last index of list and for step it is 1

So **[ : stop]** will slice list from starting till stop index and **[start : ]** will slice list from start index till end Negative value of steps shows right to left traversal instead of left to right traversal that is why **[ : -1]** prints list in reverse order.

### Example 2:

```
# Let us first create a list to demonstrate slicing
# lst contains all number from 1 to 10
lst = list(range(1, 11))
print (lst)

# below list has numbers from 2 to 5
lst1_5 = lst[1 : 5]
print (lst1_5)

# below list has numbers from 6 to 8
lst5_8 = lst[5 : 8]
print (lst5_8)

# below list has numbers from 2 to 10
lst1_ = lst[1 : ]
print (lst1_)

# below list has numbers from 1 to 5
lst_5 = lst[: 5]
print (lst_5)

# below list has numbers from 2 to 8 in step 2
```

```

lst1_8_2 = lst[1 : 8 : 2]
print (lst1_8_2)

# below list has numbers from 10 to 1
lst_rev = lst[ : : -1]
print (lst_rev)

# below list has numbers from 10 to 6 in step 2
lst_rev_9_5_2 = lst[9 : 4 : -2]
print (lst_rev_9_5_2)

```

### Output:

```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 3, 4, 5]
[6, 7, 8]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5]
[2, 4, 6, 8]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 8, 6]

```

We can use the **filter** function to filter a list based on some condition provided as a **lambda expression** as the first argument and list as the second argument, an example of which is shown below :

### Example 3:

```

import functools

# filtering odd numbers
lst = filter(lambda x : x % 2 == 1, range(1, 20))
print (list(lst))

# filtering odd square which are divisible by 5
lst = filter(lambda x : x % 5 == 0,
             [x ** 2 for x in range(1, 11) if x % 2 == 1])
print (list(lst))

# filtering negative numbers

```

```
lst = filter((lambda x: x < 0), range(-5,5))
print(list(lst))

# implementing max() function, using
print(functools.reduce(lambda a,b: a if (a > b) else b, [7, 12, 45, 100, 15]))
```

**Output:**

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
[25]
[-5, -4, -3, -2, -1]
100
```

## Python | Output using print() function

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

**Syntax:** print(value(s), sep= ' ', end = '\n', file=file, flush=flush)

**Parameters:**

- **value(s)** : Any value, and as many as you like. Will be converted to string before printed
- **sep='separator'** : (Optional) Specify how to separate the objects, if there is more than one. Default : ' '
- **end='end'** : (Optional) Specify what to print at the end. Default : '\n'
- **file** : (Optional) An object with a write method. Default : sys.stdout
- **flush** : (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

**Returns:** It returns output to the screen.

Though it is not necessary to pass arguments in the print() function, it requires an empty parenthesis at the end that tells python to execute the function rather calling it by name. Now, let's explore the optional arguments that can be used with the print() function.

### String Literals

String literals in python's print statement are primarily used to format or design how a specific string appears when printed using the print() function.

- **\n** : This string literal is used to add a new blank line while printing a statement.
- **""** : An empty quote ("" ) is used to print an empty line.

**Example:**

```
print("GeeksforGeeks \n is best for DSA Content.")
```

**Output:**

```
GeeksforGeeks
is best for DSA Content.
```

## end= " " statement

The end keyword is used to specify the content that is to be printed at the end of the execution of the print() function. By default, it is set to “\n”, which leads to the change of line after the execution of print() statement.

Example: Python print() without new line.

```
# This line will automatically add a new line before the
# next print statement

print("GeeksForGeeks is the best platform for DSA content")

# This print() function ends with "***" as set in the end argument.
print("GeeksForGeeks is the best platform for DSA content", end="***")
print("Welcome to GFG")
```

## Output:

```
GeeksForGeeks is the best platform for DSA content
GeeksForGeeks is the best platform for DSA content***Welcome to GFG
```

## flush Argument

The I/Os in python are generally buffered, meaning they are used in chunks. This is where flush comes in as it helps users to decide if they need the written content to be buffered or not. By default, it is set to false. If it is set to true, the output will be written as a sequence of characters one after the other. This process is slow simply because it is easier to write in chunks rather than writing one character at a time. To understand the use case of the flush argument in the print() function, let's take an example.

## Example:

Imagine you are building a countdown timer, which appends the remaining time to the same line every second. It would look something like below:

```
3>>>2>>>1>>>Start
```

The initial code for this would look something like below;

```
import time

count_seconds = 3

for i in reversed(range(count_seconds + 1)):
```

```
if i > 0:
    print(i, end='>>>')
    time.sleep(1)
else:
    print('Start')
```

So, the above code adds text without a trailing newline and then sleeps for one second after each text addition. At the end of the countdown, it prints Start and terminates the line. If you run the code as it is, it waits for 3 seconds and abruptly prints the entire text at once. This is a waste of 3 seconds caused due to buffering of the text chunk as shown below:

```
Microsoft Windows [Version 10.0.18363.1198]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Raju Kumar>
```

Though buffering serves a purpose, it can result in undesired effects as shown above. To counter the same issue, the flush argument is used with the print() function. Now, set the flush argument as true and again see the results.

```
import time
```

```
count_seconds = 3
```

```
for i in reversed(range(count_seconds + 1)):
```



```

if i > 0:
    print(i, end='>>>', flush = True)
    time.sleep(1)
else:
    print('Start')

```

## Output:

Video Player

00:00

00:10

## Separator

The `print()` function can accept any number of positional arguments. These arguments can be separated from each other using a “,” **separator**. These are primarily used for formatting multiple statements in a single `print()` function.

## Example:

```

b = "for"

print("Geeks", b , "Geeks")

```

## Output:

Geeks for Geeks

## file Argument

Contrary to popular belief, the `print()` function doesn't convert the messages into text on the screen. These are done by lower-level layers of code, that can read data(message) in bytes. The `print()` function is an interface over these layers, that delegates the actual printing to a stream or **file-like object**. By default, the `print()` function is bound to `sys.stdout` through the `file` argument.

[Example: Python print\(\) to file](#)

```

import io

# declare a dummy file
dummy_file = io.StringIO()

# add message to the dummy file

```

```
print('Hello Geeks!!', file=dummy_file)
```

```
# get the value from dummy file  
dummy_file.getvalue()
```

### Output:

```
'Hello Geeks!!\n'
```

[Example : Using print\(\) function in Python](#)

```
# Python 3.x program showing
```

```
# how to print data on
```

```
# a screen
```

```
# One object is passed
```

```
print("GeeksForGeeks")
```

```
x = 5
```

```
# Two objects are passed
```

```
print("x =", x)
```

```
# code for disabling the softspace feature
```

```
print('G', 'F', 'G', sep='')
```

```
# using end argument
```

```
print("Python", end='@')
```

```
print("GeeksforGeeks")
```

### Output:

```
GeeksForGeeks
```

```
x = 5
```

```
GFG
```

```
Python@GeeksforGeeks
```

[Print without newline in Python 3.x without using for loop](#)

```
# Print without newline in Python 3.x without using for loop
```

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

```
# using * symbol prints the list
# elements in a single line
print(*l)
```

```
#This code is contributed by anuragsingh1022
```

### Output:

```
1 2 3 4 5 6
```

The sep parameter when used with the [end](#) parameter it produces awesome results. Some examples by combining the sep and [end](#) parameters.

```
print('G','F', sep='', end='')
print('G')
#\n provides new line after printing the year
print('09','12','2016', sep='-', end='\n')

print('prtk','agarwal', sep='', end='@')
print('geeksforgeeks')
```

### Output:

```
GFG
09-12-2016
prtkagarwal@geeksforgeeks
```

# Python | Output Formatting

There are several ways to present the output of a program, data can be printed in a human-readable form, or written to a file for future use, or even in some other specified form. Sometimes user often wants more control over the formatting of output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with for F before the opening quotation mark or triple quotation mark.
- The [str.format\(\)](#) method of strings help a user to get a fancier Output
- Users can do all the string handling by using string slicing and concatenation operations to create any layout that the user wants. The string type has some methods that perform useful operations for padding strings to given column width.

## Formatting output using String modulo operator(%) :

The % operator can also be used for string formatting. It interprets the left argument much like a printf()-style format as in C language string to be applied to the right argument. In Python, there is no printf() function but the functionality of the ancient printf is contained in Python. To this purpose, the modulo operator % is overloaded by the string class to perform string formatting. Therefore, it is often called a string modulo (or sometimes even called modulus) operator.

The string modulo operator ( % ) is still available in Python(3.x) and the user is using it widely. But nowadays the old style of formatting is removed from the language.

```
# Python program showing how to use
# string modulo operator(%) to print
# fancier output

# print integer and float value
print("Geeks : %2d, Portal : %5.2f" % (1, 05.333))

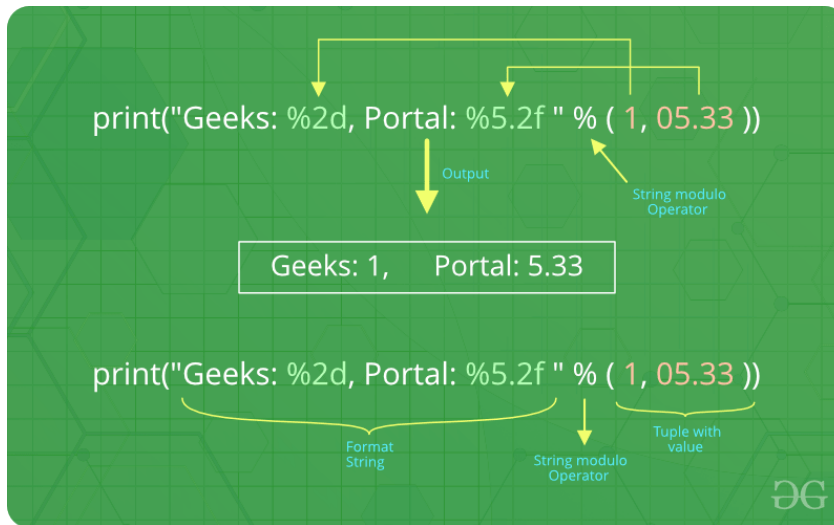
# print integer value
print("Total students : %3d, Boys : %2d" % (240, 120))

# print octal value
print("%7.3o" % (25))

# print exponential value
print("%10.3E" % (356.08977))
```

**Output :**

```
Geeks : 1, Portal : 5.33
Total students : 240, Boys : 120
031
3.561E+02
```



There are two of those in our example: “%2d” and “%5.2f”. The general syntax for a format placeholder is:

```
%[flags][width][.precision]type
```

Let’s take a look at the placeholders in our example.

- The first placeholder “%2d” is used for the first component of our tuple, i.e. the integer 1. The number will be printed with 2 characters. As 1 consists only of one digit, the output is padded with 1 leading blanks.
- The second one “%5.2f” is a format description for a float number. Like other placeholders, it is introduced with the % character. This is followed by the total number of digits the string should contain. This number includes the decimal point and all the digits, i.e. before and after the decimal point.
- Our float number 05.333 has to be formatted with 5 characters. The decimal part of the number or the precision is set to 2, i.e. the number following the “.” in our placeholder. Finally, the last character “f” of our placeholder stands for “float”.

### Formatting output using the format method :

The `format()` method was added in Python(2.6). The format method of strings requires more manual effort. Users use `{ }` to mark where a variable will be substituted and can provide detailed formatting directives, but the user also needs to provide the information to be formatted. This method lets us concatenate elements within an output through positional formatting. For Example –

#### Code 1:

```
# Python program showing
# use of format() method
```

```

# using format() method
print('I love {} for "{}!"'.format('Geeks', 'Geeks'))

# using format() method and referring
# a position of the object
print('{0} and {1}'.format('Geeks', 'Portal'))

print('{1} and {0}'.format('Geeks', 'Portal'))

# the above formatting can also be done by using f-Strings
# Although, this features work only with python 3.6 or above.

print(f"I love {'Geeks'} for \'{ 'Geeks'}!\'")

# using format() method and referring
# a position of the object
print(f"{'Geeks'} and {'Portal'}")

```

### Output :

```

I love Geeks for "Geeks!"
Geeks and Portal
Portal and Geeks

```

The brackets and characters within them (called **format fields**) are replaced with the objects passed into the format() method. A number in the brackets can be used to refer to the position of the object passed into the format() method.

### Code 2:

```

# Python program showing
# a use of format() method

# combining positional and keyword arguments
print('Number one portal is {0}, {1}, and {other}.'
      .format('Geeks', 'For', other ='Geeks'))

```

```
# using format() method with number
print("Geeks :{0:2d}, Portal :{1:8.2f}".
      format(12, 00.546))

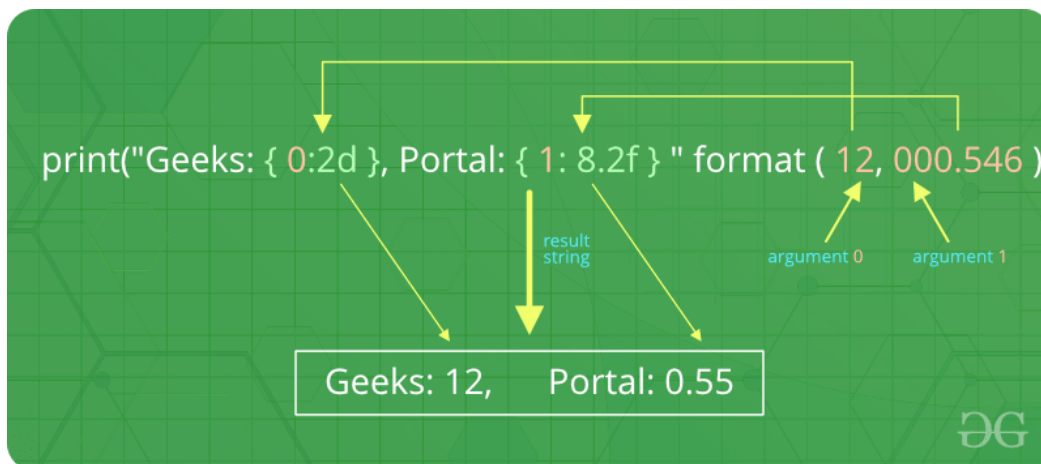
# Changing positional argument
print("Second argument: {1:3d}, first one: {0:7.2f}".
      format(47.42, 11))

print("Geeks: {a:5d}, Portal: {p:8.2f}".
      format(a = 453, p = 59.058))
```

### Output:

```
Number one portal is Geeks, For, and Geeks.
Geeks :12, Portal :    0.55
Second argument:  11, first one:   47.42
Geeks:   453, Portal:    59.06
```

The following diagram with an example usage depicts how the format method works for positional parameters:



### Code 3:

```
# Python program to
# show format () is
# used in dictionary

tab = {'geeks': 4127, 'for': 4098, 'geek': 8637678}
```

```
# using format() in dictionary
print('Geeks: {0[geeks]:d}; For: {0[for]:d}; '
      'Geeks: {0[geek]:d}'.format(tab))

data = dict(fun = "GeeksForGeeks", adj = "Portal")

# using format() in dictionary
print("I love {fun} computer {adj}".format(**data))
```

### Output:

```
Geeks: 4127; For: 4098; Geeks: 8637678
I love GeeksForGeeks computer Portal
```

### Formatting output using the String method :

This output is formatted by using string slicing and concatenation operations. The string type has some methods that help in formatting output in a fancier way. Some of method which help in formatting a output are [str.rjust\(\)](#), [str.rjust\(\)](#), [str.center\(\)](#)

```
# Python program to
# format a output using
# string() method

cstr = "I love geeksforgeeks"

# Printing the center aligned
# string with fillchr
print("Center aligned string with fillchr: ")
print(cstr.center(40, '#'))

# Printing the left aligned
# string with "-" padding
print("The left aligned string is : ")
print(cstr.ljust(40, '-'))

# Printing the right aligned string
# with "-" padding
print("The right aligned string is : ")
```



```
print (cstr.rjust(40, '-'))
```

### Output:

```
Center aligned string with fillchr:
#####I love geeksforgeeks#####

The left aligned string is :
I love geeksforgeeks-----

The right aligned string is :
-----I love geeksforgeeks
```

## f-strings in Python

[PEP 498](#) introduced a new string formatting mechanism known as *Literal String Interpolation* or more commonly as *F-strings* (because of the leading *f* character preceding the string literal). The idea behind f-strings is to make string interpolation simpler. To create an f-string, prefix the string with the letter “f”. The string itself can be formatted in much the same way that you would with [str.format\(\)](#). F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.

### Code #1 :

```
# Python3 program introducing f-string

val = 'Geeks'

print(f"{val}for{val} is a portal for {val}.")


name = 'Tushar'
age = 23

print(f"Hello, My name is {name} and I'm {age} years old.")
```

### Output :

```
GeeksforGeeks is a portal for Geeks.
Hello, My name is Tushar and I'm 23 years old.
```

### Code #2 :

```
# Prints today's date with help
```

```
# of datetime library
import datetime

today = datetime.datetime.today()
print(f"{today:%B %d, %Y}")
```

**Output :**

April 04, 2018

**Note :** F-strings are faster than the two most commonly used string formatting mechanisms, which are % formatting and str.format().

Let's see few error examples, which might occur while using f-string :

**Code #3 :** Demonstrating Syntax error.

```
answer = 456
f"Your answer is "{answer}"
```

**Code #4 :** Backslash Cannot be used in format string directly.

```
f"newline: {ord('\n')}"
```

**Output :**

```
Traceback (most recent call last):
  Python Shell, prompt 29, line 1
Syntax Error: f-string expression part cannot include a backslash: , line
1, pos 0
```

**But the documentation points out that we can put the backslash into a variable as a workaround though :**

```
newline = ord('\n')

f"newline: {newline}"
```

**Output :**

newline: 10

# Python String format() Method

- Difficulty Level : [Easy](#)
- Last Updated : 14 Oct, 2021

**Python format() function** has been introduced for handling complex string formatting more efficiently. This method of the built-in string class provides functionality for complex variable substitutions and value formatting. This new formatting technique is regarded as more elegant. The general syntax of format() method is `string.format(var1, var2,...)`

## Using a Single Formatter

Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces { } into a string and calling the `str.format()`. The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.

**Syntax :** { } .format(value)

### Parameters :

**(value) :** Can be an integer, floating point numeric constant, string, characters or even variables.

**Returntype :** Returns a formatted string with the value passed as parameter in the placeholder position.

### Example 1: Simple demonstration of format()

```
# Python3 program to demonstrate
# the str.format() method

# using format option in a simple string
print("{} , A computer science portal for geeks."
      .format("GeeksforGeeks"))

# using format option for a
# value stored in a variable
str="This article is written in {}"
print(str.format("Python"))

# formatting a string using a numeric constant
print("Hello, I am {} years old !".format(18))
```

## Output :

```
GeeksforGeeks, A computer science portal for geeks.  
This article is written in Python  
Hello, I am 18 years old!
```

## Using Multiple Formatters

Multiple pairs of curly braces can be used while formatting the string. Let's say if another variable substitution is needed in the sentence, can be done by adding a second pair of curly braces and passing a second value into the method. Python will replace the placeholders with values in **order**.

**Syntax :** { } { } .format(value1, value2)

**Parameters :** (value1, value2) : Can be integers, floating point numeric constants, strings, characters and even variables. Only difference is, the number of values passed as parameters in format() method must be equal to the number of placeholders created in the string.

## Errors and Exceptions :

**IndexError :** Occurs when string has an extra placeholder, and we didn't pass any value for it in the format() method. Python usually assigns the placeholders with default index in order like 0, 1, 2, 3.... to access the values passed as parameters. So when it encounters a placeholder whose index doesn't have any value passed inside as parameter, it throws IndexError.

### Example 2: Python String format() Method IndexError

```
# Python program demonstrating Index error  
  
# Number of placeholders are four but  
# there are only three values passed  
  
# parameters in format function.  
my_string = "{} , is a {} {} science portal for {}"  
  
print(my_string.format("GeeksforGeeks", "computer", "geeks"))
```

## Output :

```
IndexError: tuple index out of range
```

### Example 3: Python String format() with multiple placeholders

```
# Python program using multiple place  
# holders to demonstrate str.format() method
```

```
# Multiple placeholders in format() function
my_string = "{} , is a {} science portal for {}"
print(my_string.format("GeeksforGeeks", "computer", "geeks"))

# different datatypes can be used in formatting
print("Hi ! My name is {} and I am {} years old"
      .format("User", 19))

# The values passed as parameters
# are replaced in order of their entry
print("This is {} {} {} {}"
      .format("one", "two", "three", "four"))
```

## Output :

```
GeeksforGeeks, is a computer science portal for geeks
Hi! My name is User and I am 19 years old
This is one two three four
```

## Formatting Strings using Escape Sequences

You can use two or more specially designated characters within a string to format a string or perform a command. These characters are called escape sequences. An Escape sequence in Python starts with a backslash (\). For example, \n is an escape sequence in which the common meaning of the letter n is literally escaped and given an alternative meaning – a new line.

Escape sequence	Description	Example
\n	Breaks the string into a new line	print('I designed this rhyme to explain in due time\nAll I know')
\t	Adds a horizontal tab	print('Time is a \tvaluable thing')
\\	Prints a backslash	print('Watch it fly by\\as the pendulum swings')
\'	Prints a single quote	print('It doesn\'t even matter how hard you try')
\"	Prints a double quote	print('It is so\'unreal\'')
\a	makes a sound like a bell	print('\a')

## Formatters with Positional and Keyword Arguments

When placeholders { } are empty, Python will replace the values passed through str.format() in order.

The values that exist within the str.format() method are essentially **tuple data types** and each individual value contained in the tuple can be called by its index number, which starts with the index number 0. These index numbers can be passed into the curly braces that serve as the placeholders in the original string.

**Syntax :** {0} {1}.format(positional\_argument, keyword\_argument)

**Parameters :** (positional\_argument, keyword\_argument)

**Positional\_argument** can be integers, floating point numeric constants, strings, characters and even variables.

**Keyword\_argument** is essentially a variable storing some value, which is passed as parameter.

### Example 4:

```
# To demonstrate the use of formatters
# with positional key arguments.
```

```
# Positional arguments
# are placed in order
print("{0} love {1}!!".format("GeeksforGeeks",
                              "Geeks"))
```

```
# Reverse the index numbers with the
# parameters of the placeholders
print("{1} love {0}!!".format("GeeksforGeeks",
                              "Geeks"))
```

```
print("Every {} should know the use of {} {} programming and {}"
      .format("programmer", "Open", "Source",
              "Operating Systems"))
```

```
# Use the index numbers of the
# values to change the order that
```

```
# they appear in the string
print("Every {3} should know the use of {2} {1} programming and {0}"
      .format("programmer", "Open", "Source", "Operating Systems"))

# Keyword arguments are called
# by their keyword name
print("{gfg} is a {0} science portal for {1}"
      .format("computer", "geeks", gfg="GeeksforGeeks"))
```

### Output :

GeeksforGeeks love Geeks!!

Geeks love GeeksforGeeks!!

Every programmer should know the use of Open Source programming and Operating Systems

Every Operating Systems should know the use of Source Open programming and programmer

GeeksforGeeks is a computer science portal for geeks

### Type Specifying

More parameters can be included within the curly braces of our syntax. Use the format code syntax **{field\_name: conversion}**, where *field\_name* specifies the index number of the argument to the str.format() method, and conversion refers to the conversion code of the data type.

**Example: %s – string conversion via str() prior to formatting**

```
print("%20s" % ('geeksforgeeks', ))
print("%-20s" % ('Interngeeks', ))
print("%.5s" % ('Interngeeks', ))
```

### Output:

```
          geeksforgeeks
Interngeeks
Inter
```

**Example: %c– character**

```
type = 'bug'
```

```

result = 'troubling'

print('I wondered why the program was %s me. Then\
it dawned on me it was a %s .' %
      (result, type))

```

### Output:

I wondered why the program was me troubling me. Then it dawned on me it was a bug.

**Example: %i signed decimal integer and %d signed decimal integer(base-10)**

```
match = 12000
```

```

site = 'amazon'

print("%s is so useful. I tried to look\
up mobile and they had a nice one that cost %d rupees." % (site, match))

```

### Output:

amazon is so useful. I tried to look up mobiles and they had a nice one that cost 12000 rupees

### Some another useful Type Specifying

- **%u** unsigned decimal integer
- **%o** octal integer
- **f** – floating point display
- **b** – binary
- **o** – octal
- **%x** – hexadecimal with lowercase letters after 9
- **%X** – hexadecimal with uppercase letters after 9
- **e** – exponent notation

You can also specify formatting symbols. The only change is using a colon (:) instead of %. For example, instead of %s use {:s} and instead of %d use (:d}

**Syntax :** String {field\_name:conversion} Example.format(value)

### Errors and Exceptions :

**ValueError :** Error occurs during type conversion in this method.

### Example 5:

```

# Demonstrate ValueError while
# doing forced type-conversions

```



```

# When explicitly converted floating-point
# values to decimal with base-10 by 'd'
# type conversion we encounter Value-Error.
print("The temperature today is {0:d} degrees outside !"
      .format(35.567))

# Instead write this to avoid value-errors
''' print("The temperature today is {0:.0f} degrees outside !"
        .format(35.567))'''

```

## Output :

ValueError: Unknown format code 'd' for object of type 'float'

### Example 6 :

```

# Convert base-10 decimal integers
# to floating point numeric constants
print("This site is {0:f}% securely {1}!!".
      format(100, "encrypted"))

# To limit the precision
print("My average of this {0} was {1:.2f}%"
      .format("semester", 78.234876))

# For no decimal places
print("My average of this {0} was {1:.0f}%"
      .format("semester", 78.234876))

# Convert an integer to its binary or
# with other different converted bases.
print("The {0} of 100 is {1:b}"
      .format("binary", 100))

print("The {0} of 100 is {1:o}"
      .format("octal", 100))

```

## Output :

```
This site is 100.000000% securely encrypted!!
My average of this semester was 78.23%
My average of this semester was 78%
The binary of 100 is 1100100
The octal of 100 is 144
```

## Padding Substitutions or Generating Spaces

### Example 7: Demonstration of spacing when strings are passed as parameters

By default, strings are left-justified within the field, and numbers are right-justified. We can modify this by placing an alignment code just following the colon.

```
<  : left-align text in the field
^   : center text in the field
>   : right-align text in the field
# To demonstrate spacing when

# strings are passed as parameters

print("{0:4}, is the computer science portal for {1:8}!"
      .format("GeeksforGeeks", "geeks"))


# To demonstrate spacing when numeric
# constants are passed as parameters.
print("It is {0:5} degrees outside !"
      .format(40))


# To demonstrate both string and numeric
# constants passed as parameters
print("{0:4} was founded in {1:16}!"
      .format("GeeksforGeeks", 2009))


# To demonstrate aligning of spaces
print("{0:^16} was founded in {1:<4}!"
      .format("GeeksforGeeks", 2009))


print("{:*^20s}".format("Geeks"))
```

## Output :

```

GeeksforGeeks, is the computer science portal for geeks    !
It is    40 degrees outside!
GeeksforGeeks was founded in    2009!
    GeeksforGeeks    was founded in 2009 !
*****Geeks*****

```

## Applications

Formatters are generally used to Organize Data. Formatters can be seen in their best light when they are being used to organize a lot of data in a visual way. If we are showing databases to users, using formatters to increase field size and modify alignment can make the output more readable.

### Example 8: To demonstrate the organization of large data using format()

```

# which prints out i, i ^ 2, i ^ 3,

# i ^ 4 in the given range


# Function prints out values
# in an unorganized manner
def unorganized(a, b):
    for i in range(a, b):
        print(i, i**2, i**3, i**4)


# Function prints the organized set of values
def organized(a, b):
    for i in range(a, b):

        # Using formatters to give 6
        # spaces to each set of values
        print("{:6d} {:6d} {:6d} {:6d}"
              .format(i, i ** 2, i ** 3, i ** 4))


# Driver Code
n1 = int(input("Enter lower range :-\n"))
n2 = int(input("Enter upper range :-\n"))

print("-----Before Using Formatters-----")

# Calling function without formatters

```

```

unorganized(n1, n2)

print()

print("-----After Using Formatters-----")

print()

# Calling function that contains
# formatters to organize the data
organized(n1, n2)

```

## Output :

```

Enter lower range :-
3
Enter upper range :-
10
-----Before Using Formatters-----
3 9 27 81
4 16 64 256
5 25 125 625
6 36 216 1296
7 49 343 2401
8 64 512 4096
9 81 729 6561

-----After Using Formatters-----

```

3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561

## Using a dictionary for string formatting

Using a dictionary to unpack values into the placeholders in the string that needs to be formatted. We basically use `**` to unpack the values. This method can be useful in string substitution while preparing an SQL query.

```

introduction = 'My name is {first_name} {middle_name} {last_name} AKA the {aka}.'

full_name = {
    'first_name': 'Tony',
    'middle_name': 'Howard',
    'last_name': 'Stark',

```

```

        'aka': 'Iron Man',
    }

# Notice the use of "*" operator to unpack the values.
print(introduction.format(**full_name))

```

### Output:

My name is Tony Howard Stark AKA the Iron Man.

### Python format() with list

Given a list of float values, the task is to truncate all float values to 2-decimal digits. Let's see the different methods to do the task.

```

# Python code to truncate float
# values to 2 decimal digits.

# List initialization
Input = [100.7689454, 17.232999, 60.98867, 300.83748789]

# Using format
Output = [' {:.2f}'.format(elem) for elem in Input]

# Print output
print(Output)

```

### Output:

```
['100.77', '17.23', '60.99', '300.84']
```

## Python String | ljust(), rjust(), center()

- Difficulty Level : [Medium](#)
- Last Updated : 02 Feb, 2018

String alignment is frequently used in many day-day applications. Python in its language offers several functions that helps to align string. Also, offers a way to add user specified padding instead of blank space.

These functions are :

```
str.ljust(s, width[, fillchar])
str.rjust(s, width[, fillchar])
str.center(s, width[, fillchar])
```

These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least width characters wide, created by padding the string *s* with the character *fillchar* (default is a space) until the given width on the right, left or both sides. The string is never truncated.

### **center()**

This function ***center aligns*** the string according to the width specified and fills remaining space of line with blank space if ‘*fillchr*’ argument is not passed.

#### **Syntax :**

center( len, fillchr )

#### **Parameters :**

**len :** The width of string to expand it.

**fillchr (optional):** The character to fill in remaining space.

#### **Return Value :**

The resultant center aligned string expanding the given width.

```
# Python3 code to demonstrate
# the working of center()

cstr = "I love geeksforgeeks"

# Printing the original string
print("The original string is : \n", cstr, "\n")

# Printing the center aligned string
print("The center aligned string is : ")
print(cstr.center(40), "\n")

# Printing the center aligned
# string with fillchr
print("Center aligned string with fillchr: ")
print(cstr.center(40, '#'))
```

### Output :

```
The original string is :  
I love geeksforgeeks
```

```
The center aligned string is :  
    I love geeksforgeeks
```

```
Center aligned string with fillchr:  
#####I love geeksforgeeks#####
```

### **ljust()**

This function ***left aligns*** the string according to the width specified and fills remaining space of line with blank space if ‘*fillchr*’ argument is not passed.

### **Syntax :**

```
ljust( len, fillchr )
```

### **Parameters :**

**len :** The width of string to expand it.

**fillchr (optional):** The character to fill in remaining space.

### **Return Value :**

The resultant left aligned string expanding the given width.

```
# Python3 code to demonstrate  
# the working of  ljust()  
  
lstr = "I love geeksforgeeks"  
  
# Printing the original string  
print ("The original string is : \n", lstr, "\n")  
  
# Printing the left aligned  
# string with "-" padding  
print ("The left aligned string is : ")  
print (lstr.ljust(40, '-'))
```

### Output :

```
The original string is :  
I love geeksforgeeks
```

The left aligned string is :  
I love geeksforgeeks-----

### **rjust()**

This function **right aligns** the string according to the width specified and fills remaining space of line with blank space if '*fillchr*' argument is not passed.

#### **Syntax :**

rjust( len, fillchr )

#### **Parameters :**

**len :** The width of string to expand it.

**fillchr (optional) :** The character to fill in remaining space.

#### **Return Value :**

The resultant right aligned string expanding the given width.

```
# Python3 code to demonstrate
# the working of rjust()

rstr = "I love geeksforgeeks"

# Printing the original string
print("The original string is : \n", rstr, "\n")

# Printing the right aligned string
# with "-" padding
print("The right aligned string is : ")
print(rstr.rjust(40, '-'))
```

#### **Output :**

```
The original string is :
I love geeksforgeeks

The right aligned string is :
-----I love geeksforgeeks
```

## **Bitwise Operators**

[Bitwise operators](#) act on bits and perform the bit-by-bit operations. These are used to operate on binary numbers.



Operator	Description	Syntax
&	Bitwise AND	x & y
	Bitwise OR	x   y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x >>
<<	Bitwise left shift	x <<

#### Example: Bitwise Operators in Python

```
# Examples of Bitwise operators
```

```
a = 10
```

```
b = 4
```

```
# Print bitwise AND operation
```

```
print(a & b)
```

```
# Print bitwise OR operation
```

```
print(a | b)
```

```
# Print bitwise NOT operation
```

```
print(~a)
```

```
# print bitwise XOR operation
```

```
print(a ^ b)
```

```
# print bitwise right shift operation
```

```
print(a >> 2)
```

```
# print bitwise left shift operation
```

```
print(a << 2)
```

#### Output

```
0
14
-11
14
2
40
```

## Operator Associativity

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

### Example: Operator Associativity

```
# Examples of Operator Associativity
```

```
# Left-right associativity
# 100 / 10 * 10 is calculated as
# (100 / 10) * 10 and not
# as 100 / (10 * 10)
print(100 / 10 * 10)
```

```
# Left-right associativity
# 5 - 2 + 3 is calculated as
# (5 - 2) + 3 and not
# as 5 - (2 + 3)
print(5 - 2 + 3)
```

```
# left-right associativity
print(5 - (2 + 3))
```

```
# right-left associativity
# 2 ** 3 ** 2 is calculated as
# 2 ** (3 ** 2) and not
# as (2 ** 3) ** 2
print(2 ** 3 ** 2)
```

### Output

```
100.0
6
0
512
```

- **Simple Method to use ternary operator:**

```
# Program to demonstrate conditional operator
a, b = 10, 20
```

```
# Copy value of a in min if a < b else copy b
min=a if a < b else b

print(min)
```

### Output:

```
10
```

- **Direct Method by using tuples, Dictionary, and lambda**

```
# Python program to demonstrate ternary operator
a, b =10, 20

# Use tuple for selecting an item
# (if_test_false,if_test_true)[test]
# if [a<b] is true it return 1, so element with 1 index will print
# else if [a<b] is false it return 0, so element with 0 index will print
print( (b, a) [a < b] )

# Use Dictionary for selecting an item
# if [a < b] is true then value of True key will print
# elif [a<b] is false then value of False key will print
print({True: a, False: b} [a < b])

# lambda is more efficient than above two methods
# because in lambda we are assure that
# only one expression will be evaluated unlike in
# tuple and Dictionary
print((lambda: b, lambda: a)[a < b]())
```

### Output:

```
10
10
10
```

- **Ternary operator can be written as nested if-else:**

```
# Python program to demonstrate nested ternary operator
```

```
a, b = 10, 20
```

```
print("Both a and b are equal" if a == b else "a is greater than b"
      if a > b else "b is greater than a")
```

The above approach can be written as:

```
# Python program to demonstrate nested ternary operator
```

```
a, b = 10, 20
```

```
if a != b:
    if a > b:
        print("a is greater than b")
    else:
        print("b is greater than a")
else:
    print("Both a and b are equal")
```

### Output:

```
b is greater than a
```

- To use print function in ternary operator be like:-

Example: Find the Larger number among 2 using ternary operator in python3

```
a=5
```

```
b=7
```

```
# [statement_on_True] if [condition] else [statement_on_false]
```

```
print(a,"is greater") if (a>b) else print(b,"is Greater")
```

### Output:

```
7 is Greater
```

### Important Points:

- First the given condition is evaluated ( $a < b$ ), then either a or b is returned based on the Boolean value returned by the condition

- Order of the arguments in the operator is different from other languages like C/C++ (See [C/C++ ternary operators](#)).
- Conditional expressions have the lowest priority amongst all Python operations.

### Method used prior to 2.5 when the ternary operator was not present

In an expression like the one given below, the interpreter checks for the expression if this is true then `on_true` is evaluated, else the `on_false` is evaluated.

### Syntax :

```
'''When condition becomes true, expression [on_false]
is not executed and value of "True and [on_true]"
is returned. Else value of "False or [on_false]"
is returned.
Note that "True and x" is equal to x.
And "False or x" is equal to x. '''
[expression] and [on_true] or [on_false]
```

### Example :

```
# Program to demonstrate conditional operator
a, b = 10, 20

# If a is less than b, then a is assigned
# else b is assigned (Note : it doesn't
# work if a is 0.
min = a < b and a or b

print(min)
```

### Output:

```
10
```

**Note :** The only drawback of this method is that **on\_true must not be zero or False**. If this happens `on_false` will be evaluated always. The reason for that is if the expression is true, the interpreter will check for the `on_true`, if that will be zero or false, that will force the interpreter to check for `on_false` to give the final result of the whole expression.

# Operator Overloading in Python

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```
# Python program to show use of
# + operator for different purposes.

print(1 + 2)

# concatenate two strings
print("Geeks"+"For")

# Product two numbers
print(3 * 4)

# Repeat the String
print("Geeks"*4)
```

## Output:

```
3
GeeksFor
12
GeeksGeeksGeeksGeeks
```

## How to overload the operators in Python?

Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method `__add__` is automatically invoked in which the operation for + operator is defined.

## Overloading binary + operator in Python :

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class

and operators work according to that behavior defined in methods. When we use + operator, the magic method `__add__` is automatically invoked in which the operation for + operator is defined. There by changing this magic method's code, we can give extra meaning to the + operator.

#### **Code 1:**

```
# Python Program illustrate how
# to overload an binary + operator

class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a + o.a

ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)
```

#### **Output :**

```
3
GeeksFor
```

#### **Code 2:**

```
# Python Program to perform addition
# of two complex numbers using binary
# + operator overloading.

class complex:
    def __init__(self, a, b):
```

```

        self.a = a

        self.b = b

    # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)

```

### **Output :**

```
(3, 5)
```

### **Overloading comparison operators in Python :**

```

# Python program to overload
# a comparison operators

class A:
    def __init__(self, a):
        self.a = a

    def __gt__(self, other):
        if(self.a>other.a):
            return True
        else:
            return False

ob1 = A(2)
ob2 = A(3)

if(ob1>ob2):
    print("ob1 is greater than ob2")
else:
    print("ob2 is greater than ob1")

```



## Output :

```
ob2 is greater than ob1
```

## Overloading equality and less than operators :

```
# Python program to overload equality
# and less than operators

class A:
    def __init__(self, a):
        self.a = a
    def __lt__(self, other):
        if(self.a<other.a):
            return "ob1 is lessthan ob2"
        else:
            return "ob2 is less than ob1"
    def __eq__(self, other):
        if(self.a == other.a):
            return "Both are equal"
        else:
            return "Not equal"

ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)

ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)
```

## Output :

```
ob1 is lessthan ob2
Not equal
```

*Python magic methods or special functions for operator overloading*

Binary Operators:

Operator	Magic Method
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other)</code>
>>	<code>__rshift__(self, other)</code>
<<	<code>__lshift__(self, other)</code>
&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

Comparison Operators :

Operator	Magic Method
<	<code>__LT__(SELF, OTHER)</code>
>	<code>__GT__(SELF, OTHER)</code>
<=	<code>__LE__(SELF, OTHER)</code>
>=	<code>__GE__(SELF, OTHER)</code>
==	<code>__EQ__(SELF, OTHER)</code>
!=	<code>__NE__(SELF, OTHER)</code>

Assignment Operators :

Operator	Magic Method
-=	<code>__ISUB__(SELF, OTHER)</code>
+=	<code>__IADD__(SELF, OTHER)</code>
*=	<code>__IMUL__(SELF, OTHER)</code>
/=	<code>__IDIV__(SELF, OTHER)</code>
//=	<code>__IFLOORDIV__(SELF, OTHER)</code>

```

%=      __IMOD__(SELF, OTHER)

**=     __IPOW__(SELF, OTHER)

>>=    __IRSHIFT__(SELF, OTHER)

<<=    __ILSHIFT__(SELF, OTHER)

&=      __IAND__(SELF, OTHER)

|=      __IOR__(SELF, OTHER)

^=      __IXOR__(SELF, OTHER)

```

#### *Unary Operators :*

Operator	Magic Method
----------	--------------

-	__NEG__(SELF, OTHER)
+	__POS__(SELF, OTHER)
~	__INVERT__(SELF, OTHER)

## Any All in Python

Any and All are two built ins provided in python used for successive And/Or.

### **Any**

Returns true if any of the items is True. It returns False if empty or all are false. Any can be thought of as a sequence of OR operations on the provided iterables.

It short circuit the execution i.e. stop the execution as soon as the result is known.

**Syntax :** any(list of iterables)

```

# Since all are false, false is returned
print (any([False, False, False, False]))

# Here the method will short-circuit at the
# second item (True) and will return True.
print (any([False, True, False, False]))

# Here the method will short-circuit at the
# first (True) and will return True.
print (any([True, False, False, False]))

```

**Output :**

```
False
True
True
```

## All

Returns true if all of the items are True (or if the iterable is empty). All can be thought of as a sequence of AND operations on the provided iterables. It also short circuit the execution i.e. stop the execution as soon as the result is known.

### Syntax : all(list of iterables)

```
# Here all the iterables are True so all
# will return True and the same will be printed
print(all([True, True, True, True]))
```

```
# Here the method will short-circuit at the
# first item (False) and will return False.
print(all([False, True, True, False]))
```

```
# This statement will return False, as no
# True is found in the iterables
print(all([False, False, False]))
```

### Output :

```
True
False
False
```

### Practical Examples

```
# This code explains how can we
# use 'any' function on list
list1 = []
list2 = []

# Index ranges from 1 to 10 to multiply
for i in range(1,11):
    list1.append(4*i)
```

```
# Index to access the list2 is from 0 to 9
for i in range(0,10):
    list2.append(list1[i]%5==0)

print('See whether at least one number is divisible by 5 in list 1=>')
print(any(list2))
```

### **Output:**

```
See whether at least one number is divisible by 5 in list 1=>
True
# Illustration of 'all' function in python 3
```

```
# Take two lists
list1=[]
list2=[]

# All numbers in list1 are in form: 4*i-3
for i in range(1,21):
    list1.append(4*i-3)

# list2 stores info of odd numbers in list1
for i in range(0,20):
    list2.append(list1[i]%2==1)

print('See whether all numbers in list1 are odd =>')
print(all(list2))
```

### **Output:**

```
See whether all numbers in list1 are odd =>
True
```

	any	all
All Truthy values	True	True
All Falsy values	False	False
One Truthy value(all others are Falsy)	True	False
One Falsy value(all others are Truthy)	True	False
Empty Iterable	False	True

## Operator Functions in Python | Set 1

Python has predefined functions for many mathematical, logical, relational, bitwise etc operations under the module “operator”. Some of the basic functions are covered in this article.

**1. add(a, b) :-** This functions returns **addition** of the given arguments.  
Operation – **a + b.**

**2. sub(a, b) :-** This functions returns **difference** of the given arguments.  
Operation – **a – b.**

**3. mul(a, b) :-** This functions returns **product** of the given arguments.  
Operation – **a \* b.**

```
# Python code to demonstrate working of
```

```
# add(), sub(), mul()
```

```
# importing operator module
```

```
import operator
```

```
# Initializing variables
```

```
a = 4
```

```
b = 3
```

```
# using add() to add two numbers
print("The addition of numbers is :",end="");
print(operator.add(a, b))

# using sub() to subtract two numbers
print("The difference of numbers is :",end="");
print(operator.sub(a, b))

# using mul() to multiply two numbers
print("The product of numbers is :",end="");
print(operator.mul(a, b))
```

Output:

```
The addition of numbers is:7
The difference of numbers is :1
The product of numbers is:12
```

**4. `truediv(a,b)` :-** This function returns **division** of the given arguments.  
Operation – **`a / b`**.

**5. `floordiv(a,b)` :-** This function also returns division of the given arguments. But the value is floored value i.e. **returns greatest small integer**.  
Operation – **`a // b`**.

**6. `pow(a,b)` :-** This function returns **exponentiation** of the given arguments.  
Operation – **`a ** b`**.

**7. `mod(a,b)` :-** This function returns **modulus** of the given arguments.  
Operation – **`a % b`**.

```
# Python code to demonstrate working of
# truediv(), floordiv(), pow(), mod()

# importing operator module
import operator
```

```
# Initializing variables
a = 5
```

```

b = 2

# using truediv() to divide two numbers
print("The true division of numbers is : ",end="");
print(operator.truediv(a,b))

# using floordiv() to divide two numbers
print("The floor division of numbers is : ",end="");
print(operator.floordiv(a,b))

# using pow() to exponentiate two numbers
print("The exponentiation of numbers is : ",end="");
print(operator.pow(a,b))

# using mod() to take modulus of two numbers
print("The modulus of numbers is : ",end="");
print(operator.mod(a,b))

```

### Output:

```

The true division of numbers is: 2.5
The floor division of numbers is: 2
The exponentiation of numbers is: 25
The modulus of numbers is: 1

```

**8. lt(a, b) :-** This function is used to **check if a is less than b or not**. Returns true if a is less than b, else returns false.  
Operation – **a < b**.

**9. le(a, b) :-** This function is used to **check if a is less than or equal to b or not**. Returns true if a is less than or equal to b, else returns false.  
Operation – **a <= b**.

**10. eq(a, b) :-** This function is used to **check if a is equal to b or not**. Returns true if a is equal to b, else returns false.  
Operation – **a == b**.

```

# Python code to demonstrate working of
# lt(), le() and eq()

# importing operator module

```



```

import operator

# Initializing variables
a = 3

b = 3

# using lt() to check if a is less than b
if(operator.lt(a,b)):
    print("3 is less than 3")
else: print("3 is not less than 3")

# using le() to check if a is less than or equal to b
if(operator.le(a,b)):
    print("3 is less than or equal to 3")
else: print("3 is not less than or equal to 3")

# using eq() to check if a is equal to b
if(operator.eq(a,b)):
    print("3 is equal to 3")
else: print("3 is not equal to 3")

```

**Output:**

```

3 is not less than 3
3 is less than or equal to 3
3 is equal to 3

```

**11. gt(a,b) :-** This function is used to **check if a is greater than b or not**. Returns true if a is greater than b, else returns false.

Operation – **a > b**.

**12. ge(a,b) :-** This function is used to **check if a is greater than or equal to b or not**.

Returns true if a is greater than or equal to b, else returns false.

Operation – **a >= b**.

**13. ne(a,b) :-** This function is used to **check if a is not equal to b or is equal**. Returns true if a is not equal to b, else returns false.

Operation – **a != b**.

```
# Python code to demonstrate working of
# gt(), ge() and ne()

# importing operator module
import operator

# Initializing variables
a = 4

b = 3

# using gt() to check if a is greater than b
if (operator.gt(a,b)):
    print ("4 is greater than 3")
else: print ("4 is not greater than 3")

# using ge() to check if a is greater than or equal to b
if (operator.ge(a,b)):
    print ("4 is greater than or equal to 3")
else: print ("4 is not greater than or equal to 3")

# using ne() to check if a is not equal to b
if (operator.ne(a,b)):
    print ("4 is not equal to 3")
else: print ("4 is equal to 3")
```

**Output:**

```
4 is greater than 3
4 is greater than or equal to 3
4 is not equal to 3
```

## Operator Functions in Python | Set 2

More functions are discussed in this article.

**1. setitem(ob, pos, val) :-** This function is used to **assign** the value at a **particular position** in the container.

Operation – **ob[pos] = val**

**2. delitem(ob, pos) :-** This function is used to **delete** the value at a **particular position** in the container.

Operation – **del ob[pos]**

**3. getitem(ob, pos) :-** This function is used to **access** the value at a **particular position** in the container.

Operation – **ob[pos]**

```
# Python code to demonstrate working of
# setitem(), delitem() and getitem()

# importing operator module
import operator

# Initializing list
li = [1, 5, 6, 7, 8]

# printing original list
print ("The original list is : ",end="")
for i in range(0,len(li)):
    print (li[i],end=" ")

print ("\r")

# using setitem() to assign 3 at 4th position
operator.setitem(li,3,3)

# printing modified list after setitem()
print ("The modified list after setitem() is : ",end="")
for i in range(0,len(li)):
    print (li[i],end=" ")

print ("\r")
```

```

# using delitem() to delete value at 2nd index
operator.delitem(li,1)

# printing modified list after delitem()
print("The modified list after delitem() is : ",end="")
for i in range(0,len(li)):
    print(li[i],end=" ")

print("\r")

# using getitem() to access 4th element
print("The 4th element of list is : ",end="")
print(operator.getitem(li,3))

```

Output:

```

The original list is : 1 5 6 7 8
The modified list after setitem() is : 1 5 6 3 8
The modified list after delitem() is : 1 6 3 8
The 4th element of list is : 8

```

**4. setitem(ob, slice(a,b), vals) :-** This function is used to **set the values in a particular range** in the container.

Operation – **obj[a:b] = vals**

**5. delitem(ob, slice(a,b)) :-** This function is used to **delete the values from a particular range** in the container.

Operation – **del obj[a:b]**

**6. getitem(ob, slice(a,b)) :-** This function is used to **access the values in a particular range** in the container.

Operation – **obj[a:b]**

```

# Python code to demonstrate working of
# setitem(), delitem() and getitem()

# importing operator module
import operator

```

```
# Initializing list
li = [1, 5, 6, 7, 8]

# printing original list
print("The original list is : ",end="")
for i in range(0,len(li)):
    print (li[i],end=" ")

print ("\r")

# using setitem() to assign 2,3,4 at 2nd,3rd and 4th index
operator.setitem(li,slice(1,4),[2,3,4])

# printing modified list after setitem()
print("The modified list after setitem() is : ",end="")
for i in range(0,len(li)):
    print (li[i],end=" ")

print ("\r")

# using delitem() to delete value at 3rd and 4th index
operator.delitem(li,slice(2,4))

# printing modified list after delitem()
print("The modified list after delitem() is : ",end="")
for i in range(0,len(li)):
    print (li[i],end=" ")

print ("\r")

# using getitem() to access 1st and 2nd element
print("The 1st and 2nd element of list is : ",end="")
print (operator.getitem(li,slice(0,2)))
```

Output:

```
The original list is : 1 5 6 7 8
The modified list after setitem() is : 1 2 3 4 8
The modified list after delitem() is : 1 2 8
The 1st and 2nd element of list is : [1, 2]
```

**7. concat(obj1,obj2) :-** This function is used to **concatenate** two containers.

Operation – **obj1 + obj2**

**8. contains(obj1,obj2) :-** This function is used to **check if obj2 is present in obj1**.

Operation – **obj2 in obj1**

```
# Python code to demonstrate working of
# concat() and contains()

# importing operator module
import operator

# Initializing string 1
s1 = "geeksfor"

# Initializing string 2
s2 = "geeks"

# using concat() to concatenate two strings
print("The concatenated string is : ",end="")
print(operator.concat(s1,s2))

# using contains() to check if s1 contains s2
if(operator.contains(s1,s2)):
    print("geeksfor contains geeks")
else: print("geeksfor does not contain geeks")
```

Output:

```
The concatenated string is : geeksforgeeks
```

geeksfor contains geeks

**9. and\_(a,b) :-** This function is used to compute **bitwise and** of the mentioned arguments.  
Operation – **a & b**

**10. or\_(a,b) :-** This function is used to compute **bitwise or** of the mentioned arguments.  
Operation – **a | b**

**11. xor(a,b) :-** This function is used to compute **bitwise xor** of the mentioned arguments.  
Operation – **a ^ b**

**12. invert(a) :-** This function is used to compute **bitwise inversion** of the mentioned argument.  
Operation – **~ a**

```
# Python code to demonstrate working of
# and_(), or_(), xor(), invert()

# importing operator module
import operator

# Initializing a and b

a = 1

b = 0

# using and_() to display bitwise and operation
print("The bitwise and of a and b is : ",end="")
print(operator.and_(a,b))

# using or_() to display bitwise or operation
print("The bitwise or of a and b is : ",end="")
print(operator.or_(a,b))

# using xor() to display bitwise exclusive or operation
print("The bitwise xor of a and b is : ",end="")
print(operator.xor(a,b))
```

```
# using invert() to invert value of a
operator.invert(a)

# printing modified value
print("The inverted value of a is : ",end="")
print(operator.invert(a))
```

### Output:

```
The bitwise and of a and b is : 0
The bitwise or of a and b is : 1
The bitwise xor of a and b is : 1
The inverted value of a is : -2
```

## Difference between == and is operator in Python

The Equality operator (==) compares the values of both the operands and checks for value equality. Whereas the 'is' operator checks whether both the operands refer to the same object or not (present in the same memory location).

```
# python3 code to
# illustrate the
# difference between
# == and is operator
# [] is an empty list
list1 = []
list2 = []
list3=list1

if (list1 == list2):
    print("True")
else:
    print("False")
```



```

if (list1 is list2):
    print("True")
else:
    print("False")

if (list1 is list3):
    print("True")
else:
    print("False")

list3 = list3 + list2

if (list1 is list3):
    print("True")
else:
    print("False")

```

### Output:

```

True
False
True
False

```

- The output of the **first if** the condition is “True” as both list1 and list2 are empty lists.
- **Second, if** the condition shows “False” because two empty lists are at different memory locations. Hence list1 and list2 refer to different objects. We can check it with **id()** function in python which returns the “identity” of an object.
- The output of the **third if** the condition is “True” as both list1 and list3 are pointing to the same object.
- The output of the **fourth if** the condition is “False” because the concatenation of two lists always produces a new list.

```

list1 = []
list2 = []

print(id(list1))
print(id(list2))

```

### Output:

```

139877155242696
139877155253640

```

## Membership Operators

Membership operators are operators used to validate the membership of a value. It tests for membership in a sequence, such as strings, lists, or tuples.

- **in operator:** The 'in' operator is used to check if a value exists in a sequence or not. Evaluate to true if it finds a variable in the specified sequence and false otherwise.

```
# Python program to illustrate
# Finding common member in list
# using 'in' operator
list1=[1,2,3,4,5]
list2=[6,7,8,9]
for item in list1:
    if item in list2:
        print("overlapping")
else:
    print("not overlapping")
```

### Output:

```
not overlapping
```

### The same example without using in operator:

```
# Python program to illustrate
# Finding common member in list
# without using 'in' operator

# Define a function() that takes two lists
def overlapping(list1,list2):

    c=0
    d=0
    for i in list1:
        c+=1
    for i in list2:
        d+=1
    for i in range(0,c):
```

```

        for j in range(0,d):
            if(list1[i]==list2[j]):
                return 1
        return 0
list1=[1,2,3,4,5]
list2=[6,7,8,9]
if(overlapping(list1,list2)):
    print("overlapping")
else:
    print("not overlapping")

```

### Output:

not overlapping

- **‘not in’ operator**- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

```

# Python program to illustrate
# not 'in' operator
x = 24
y = 20
list = [10, 20, 30, 40, 50];

if ( x not in list ):
    print("x is NOT present in given list")
else:
    print("x is present in given list")

if ( y in list ):
    print("y is present in given list")
else:
    print("y is NOT present in given list")

```

### Output:

x is NOT present in given list  
y is present in given list

## Identity operators

In Python identity operators are used to determine whether a value is of a certain class or type. They are usually used to determine the type of data a certain variable contains. There are different identity operators such as

- **'is' operator** – Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

```
# Python program to illustrate the use
# of 'is' identity operator
x = 5
if (type(x) is int):
    print("true")
else:
    print("false")
```

### Output:

```
true
```

- **'is not' operator** – Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

```
# Python program to illustrate the
# use of 'is not' identity operator
x = 5.2
if (type(x) is not int):
    print("true")
else:
    print("false")
```

### Output:

```
true
```

# Ways to concatenate two lists in Python

Let's see how to concatenate two lists using different methods in Python. This operation is useful when we have numbers of lists of elements which needs to be processed in a similar manner.

## Method #1 : Using Naive Method

In this method, we traverse the second list and keep appending elements in the first list, so that first list would have all the elements in both lists and hence would perform the append.

```
# Python3 code to demonstrate list
# concatenation using naive method

# Initializing lists
test_list1 = [1, 4, 5, 6, 5]
test_list2 = [3, 5, 7, 2, 5]

# using naive method to concat
for i in test_list2 :
    test_list1.append(i)

# Printing concatenated list
print ("Concatenated list using naive method : "
      + str(test_list1))
```

### Output:

```
Concatenated list using naive method : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]
```

## Method #2 : Using + operator

The most conventional method to perform the list concatenation, the use of “+” operator can easily add the whole of one list behind the other list and hence perform the concatenation.

```
# Python 3 code to demonstrate list
# concatenation using + operator

# Initializing lists
test_list3 = [1, 4, 5, 6, 5]
```

```

test_list4 = [3, 5, 7, 2, 5]

# using + operator to concat
test_list3 = test_list3 + test_list4

# Printing concatenated list
print ("Concatenated list using + : "
      + str(test_list3))

```

**Output:**

```
Concatenated list using + : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]
```

### Method #3 : Using list comprehension

List comprehension can also accomplish this task of list concatenation. In this case, a new list is created, but this method is a one liner alternative to the loop method discussed above.

```

# Python3 code to demonstrate list
# concatenation using list comprehension

# Initializing lists
test_list1 = [1, 4, 5, 6, 5]
test_list2 = [3, 5, 7, 2, 5]

# using list comprehension to concat
res_list = [y for x in [test_list1, test_list2] for y in x]

# Printing concatenated list
print ("Concatenated list using list comprehension: "
      + str(res_list))

```

**Output:**

```
Concatenated list using list comprehension: [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]
```

### Method #4 : Using extend()

extend() is the function extended by lists in Python and hence can be used to perform this task. This function performs the inplace extension of first list.

```

# Python3 code to demonstrate list
# concatenation using list.extend()

# Initializing lists
test_list3 = [1, 4, 5, 6, 5]
test_list4 = [3, 5, 7, 2, 5]

# using list.extend() to concat
test_list3.extend(test_list4)

# Printing concatenated list
print ("Concatenated list using list.extend() : "
      + str(test_list3))

```

#### **Output:**

```
Concatenated list using list.extend() : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]
```

#### **Method #5 : Using \* operator**

Using \* operator, this method is the new addition to list concatenation and works only in Python 3.6+. Any no. of lists can be concatenated and returned in a new list using this operator.

```

# Python3 code to demonstrate list
# concatenation using * operator

# Initializing lists
test_list1 = [1, 4, 5, 6, 5]
test_list2 = [3, 5, 7, 2, 5]

# using * operator to concat
res_list = [*test_list1, *test_list2]

# Printing concatenated list
print ("Concatenated list using * operator : "
      + str(res_list))

```

#### **Output:**

Concatenated list using \* operator : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]

### **Method #6 : Using itertools.chain()**

`itertools.chain()` returns the iterable after chaining its arguments in one and hence does not require to store the concatenated list if only its initial iteration is required. This is useful when concatenated list has to be used just once.

```
# Python3 code to demonstrate list
# concatenation using itertools.chain()
import itertools

# Initializing lists
test_list1 = [1, 4, 5, 6, 5]
test_list2 = [3, 5, 7, 2, 5]

# using itertools.chain() to concat
res_list = list(itertools.chain(test_list1, test_list2))

# Printing concatenated list
print("Concatenated list using itertools.chain() : "
      + str(res_list))
```

### **Output:**

Concatenated list using itertools.chain() : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]



# List comprehension and ord() in Python to remove all characters other than alphabets

Given a string consisting of alphabets and other characters, remove all the characters other than alphabets and print the string so formed.

Examples:

```
Input : str = "$Gee*k;s..fo, r'Ge^eks?"
```

```
Output : GeeksforGeeks
```

We will solve this problem in python quickly using [List Comprehension](#).

**Approach :** is

1. Traverse string
2. Select characters which lie in range of [a-z] or [A-Z]
3. Print them together

## How does ord() and range() function works in python ?

- The **ord()** method returns an integer representing the Unicode code point of the given Unicode character. **For example,**

```
ord('5') = 53 and ord('A') = 65 and ord('$') = 36
```

- The **range(a,b,step)** function generates a list of elements which ranges from a inclusive to b exclusive with increment/decrement of given step.

```
# Python code to remove all characters
```

```
# other than alphabets from string
```

```
def removeAll(input):
```

```
    # Traverse complete string and separate
```

```
    # all characters which lies between [a-z] or [A-Z]
```

```
    sepChars = [char for char in input if
```

```
ord(char) in range(ord('a'),ord('z')+1,1) or ord(char) in
```

```
range(ord('A'),ord('Z')+1,1)]
```

```
    # join all separated characters
```

```
    # and print them together
```

```
    return ''.join(sepChars)
```

```
# Driver program
if __name__ == "__main__":
    input = "$Gee*k;s..fo, r'Ge^eks?"
    print (removeAll(input))
```

### Output:

GeeksforGeeks

To remove all the characters other than alphabets(a-z) && (A-Z), we just compare the character with the ASCII value and the character whose value does not lie in the range of alphabets, we remove those character using [string erase function](#).

```
# Python3 program to remove all the
# characters other than alphabets

# function to remove characters and
# pr new string
def removeSpecialCharacter(s):

    i = 0

    while i < len(s):

        # Finding the character whose
        # ASCII value fall under this
        # range
        if (ord(s[i]) < ord('A') or
            ord(s[i]) > ord('Z') and
            ord(s[i]) < ord('a') or
            ord(s[i]) > ord('z')):

            # erase function to erase
            # the character
            del s[i]
```

```

        i -= 1

    i += 1

print("".join(s))

# Driver Code
if __name__ == '__main__':
    s = "$Gee*k;s..fo, r'Ge^eks?"
    s = [i for i in s]
    removeSpecialCharacter(s)

# This code is contributed by Mohit Kumar

```

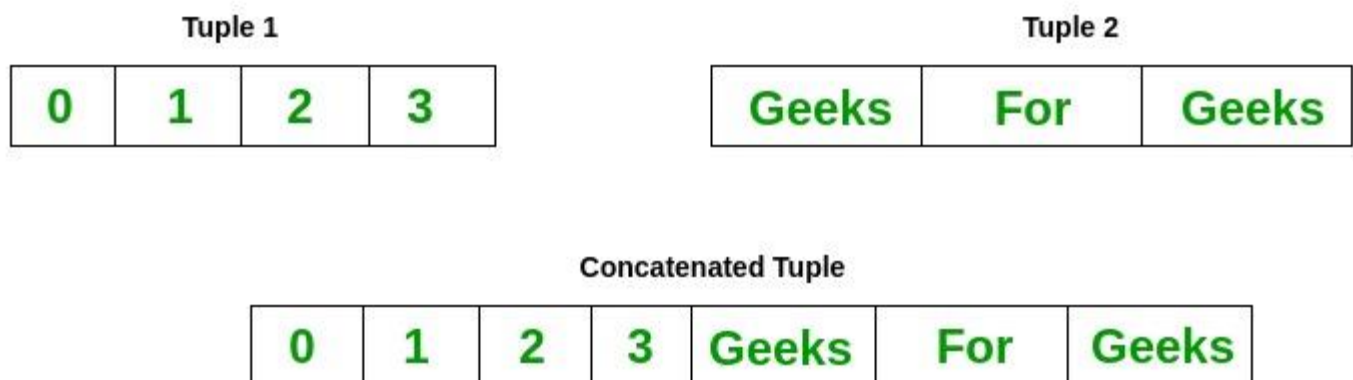
## Output:

GeeksforGeeks

## Concatenation of Tuples

Concatenation of tuple is the process of joining two or more Tuples. Concatenation is done by the use of '+' operator. Concatenation of tuples is done always from the end of the original tuple. Other arithmetic operations do not apply on Tuples.

**Note-** Only the same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.



```

# Concatenation of tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('Geeks', 'For', 'Geeks')

```

```

Tuple3 = Tuple1 + Tuple2

# Printing first Tuple
print("Tuple 1: ")
print(Tuple1)

# Printing Second Tuple
print("\nTuple2: ")
print(Tuple2)

# Printing Final Tuple
print("\nTuples after Concatenation: ")
print(Tuple3)

```

### Output:

```

Tuple 1:
(0, 1, 2, 3)

Tuple2:
('Geeks', 'For', 'Geeks')

Tuples after Concatenation:
(0, 1, 2, 3, 'Geeks', 'For', 'Geeks')

```

## Unpacking a Tuple in Python

**Python Tuples** In python [tuples](#) are used to store immutable objects. Python Tuples are very similar to lists except to some situations. Python tuples are immutable means that they can not be modified in whole program.

**Packing and Unpacking a Tuple:** In Python, there is a very powerful tuple assignment feature that assigns the right-hand side of values into the left-hand side. In another way, it is called unpacking of a tuple of values into a variable. In packing, we put values into a new tuple while in unpacking we extract those values into a single variable.

### Example 1

```

# Program to understand about
# packing and unpacking in Python

```

```

# this lines PACKS values
# into variable a
a = ("MNNIT Allahabad", 5000, "Engineering")

# this lines UNPACKS values
# of variable a
(college, student, type_ofcollege) = a

# print college name
print(college)

# print no of student
print(student)

# print type of college
print(type_ofcollege)

```

#### **Output:**

```

MNNIT Allahabad
5000
Engineering

```

**NOTE :** In unpacking of tuple number of variables on left-hand side should be equal to number of values in given tuple a.

Python uses a special syntax to pass optional arguments (\*args) for tuple unpacking. This means that there can be many number of arguments in place of (\*args) in python. All values will be assigned to every variable on the left-hand side and all remaining values will be assigned to \*args .For better understanding consider the following code.

#### **Example 2**

```

# Python3 code to study about
# unpacking python tuple using *

# first and last will be assigned to x and z
# remaining will be assigned to y
x, *y, z = (10, "Geeks ", " for ", "Geeks ", 50)

```

```

# print details
print(x)
print(y)
print(z)

# first and second will be assigned to x and y
# remaining will be assigned to z
x, y, *z = (10, "Geeks ", " for ", "Geeks ", 50)
print(x)
print(y)
print(z)

```

**Output:**

```

10
['Geeks ', ' for ', 'Geeks ']
50
10
Geeks
[' for ', 'Geeks ', 50]

```

In python tuples can be unpacked using a function in function tuple is passed and in function values are unpacked into normal variable. Consider the following code for better understanding.

**Example 3 :**

```

# Python3 code to study about
# unpacking python tuple using function

# function takes normal arguments
# and multiply them
def result(x, y):
    return x * y

# function with normal variables
print (result(10, 100))

# A tuple is created
z = (10, 100)

```

```
# Tuple is passed
# function unpacked them
```

```
print (result(*z))
```

**Output:**

```
1000
1000
```

### [Accessing a Set](#)

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

```
# Python program to demonstrate
# Accessing of elements in a set

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

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

# Checking the element
# using in keyword
print("Geeks" in set1)
```

**Output:**

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

Elements of set:
```

Geeks For

True

### Removing elements from the Set

#### *Using remove() method or discard() method*

Elements can be removed from the Set by using built-in remove() function but a KeyError arises if element doesn't exist in the set. To remove elements from a set without KeyError, use discard(), if the element doesn't exist in the set, it remains unchanged.

```
# Python program to demonstrate
# Deletion of elements in a Set

# Creating a Set
set1 = set([1, 2, 3, 4, 5, 6,
            7, 8, 9, 10, 11, 12])
print("Initial Set: ")
print(set1)

# Removing elements from Set
# using Remove() method
set1.remove(5)
set1.remove(6)
print("\nSet after Removal of two elements: ")
print(set1)

# Removing elements from Set
# using Discard() method
set1.discard(8)
set1.discard(9)
print("\nSet after Discarding two elements: ")
print(set1)

# Removing elements from Set
# using iterator method
for i in range(1, 5):
    set1.remove(i)
```



```
print("\nSet after Removing a range of elements: ")
print(set1)
```

## Output:

```
Initial Set:
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Set after Removal of two elements:
{1, 2, 3, 4, 7, 8, 9, 10, 11, 12}

Set after Discarding two elements:
{1, 2, 3, 4, 7, 10, 11, 12}

Set after Removing a range of elements:
{7, 10, 11, 12}

intersection_update() Updates the set with the intersection of itself and another
```

[isdisjoint\(\)](#) Returns True if two sets have a null intersection

[difference\\_update\(\)](#) Removes all elements of another set from this set

[discard\(\)](#) Removes an element from set if it is a member. (Do nothing if the element is not in set)

**Frozen sets** in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

If no parameters are passed, it returns an empty frozenset.

```
# Python program to demonstrate
# working of a FrozenSet

# Creating a Set
String = ('G', 'e', 'e', 'k', 's', 'F', 'o', 'r')

Fset1 = frozenset(String)
print("The FrozenSet is: ")
print(Fset1)

# To print Empty Frozen Set
# No parameter is passed
print("\nEmpty FrozenSet: ")
print(frozenset())
```

[Using popitem\(\) method](#)

The popitem() returns and removes an arbitrary element (key, value) pair from the dictionary.

```
# Creating Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
```

```
# Deleting an arbitrary key
# using popitem() function
pop_ele = Dict.popitem()
print("\nDictionary after deletion: " + str(Dict))
print("The arbitrary pair returned is: " + str(pop_ele))
```

### **Output:**

```
Dictionary after deletion: {3: 'Geeks', 'name': 'For'}
The arbitrary pair returned is: (1, 'Geeks')
```

**setdefault()**            Set dict[key]=default if key is not already in dict

**has\_key()**            Returns true if key in dictionary dict, false otherwise

**fromkeys()** Create a new dictionary with keys from seq and values set to value.

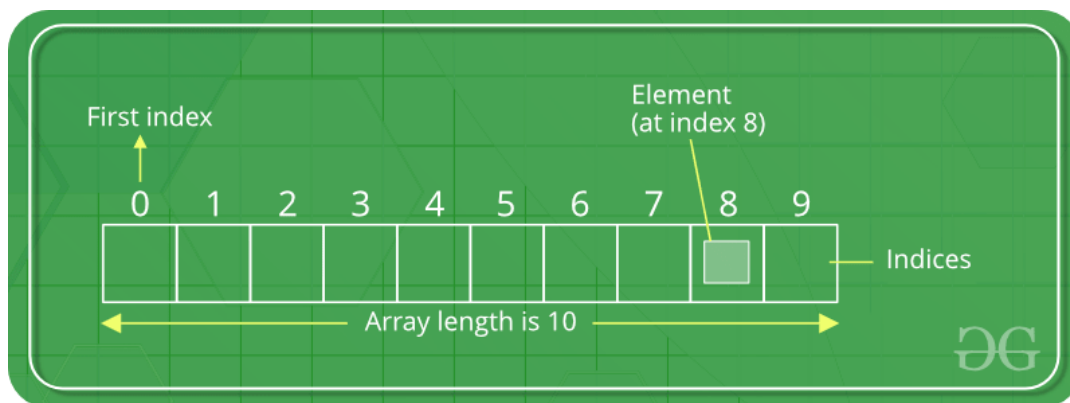
**type()**               Returns the type of the passed variable.

**cmp()**                Compares elements of both dict.

# Python Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on. **Array can be handled in Python by a module named `array`. They can be useful when we have to manipulate only a specific data type values. A user can treat [lists](#) as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the `array` module, all elements of the array must be of the same type.**



## Creating a Array

Array in Python can be created by importing `array` module. `array(data_type, value_list)` is used to create an array with data type and value list specified in its arguments.

```
# Python program to demonstrate
# Creation of Array

# importing "array" for array creations
import array as arr

# creating an array with integer type
a = arr.array('i', [1, 2, 3])

# printing original array
```

```

print ("The new created array is : ", end =" ")

for i in range (0, 3):
    print (a[i], end =" ")

print()

# creating an array with float type
b =arr.array('d', [2.5, 3.2, 3.3])

# printing original array
print ("The new created array is : ", end =" ")

for i in range (0, 3):
    print (b[i], end =" ")

```

### Output :

```

The new created array is :  1 2 3
The new created array is :  2.5 3.2 3.3

```

Some of the data types are mentioned below which will help in creating an array of different data types.

Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

### *Adding Elements to a Array*

Elements can be added to the Array by using built-in [insert\(\)](#) function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. [append\(\)](#) is also used to add the value mentioned in its arguments at the end of the array.

```
# Python program to demonstrate
# Adding Elements to a Array

# importing "array" for array creations
import array as arr

# array with int type
a = arr.array('i', [1, 2, 3])

print ("Array before insertion : ", end = " ")
for i in range (0, 3):
    print (a[i], end = " ")
print()

# inserting array using
# insert() function
a.insert(1, 4)

print ("Array after insertion : ", end = " ")
for i in (a):
    print (i, end = " ")
print()

# array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

print ("Array before insertion : ", end = " ")
```

```

for i in range (0, 3):
    print (b[i], end =" ")
print()

# adding an element using append()
b.append(4.4)

print ("Array after insertion : ", end =" ")
for i in (b):
    print (i, end =" ")
print()

```

### **Output :**

```

Array before insertion : 1 2 3
Array after insertion :  1 4 2 3
Array before insertion : 2.5 3.2 3.3
Array after insertion :  2.5 3.2 3.3 4.4

```

### *Accessing elements from the Array*

In order to access the array items refer to the index number. Use the index operator [ ] to access an item in a array. The index must be an integer.

```

# Python program to demonstrate
# accessing of element from list

# importing array module
import array as arr

# array with int type
a =arr.array('i', [1, 2, 3, 4, 5, 6])

# accessing element of array
print("Access element is: ", a[0])

```

```
# accessing element of array
print("Access element is: ", a[3])

# array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

# accessing element of array
print("Access element is: ", b[1])

# accessing element of array
print("Access element is: ", b[2])
```

### Output :

```
Access element is:  1
Access element is:  4
Access element is:  3.2
Access element is:  3.3
```

### *Removing Elements from the Array*

Elements can be removed from the array by using built-in [remove\(\)](#) function but an Error arises if element doesn't exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator is used. [pop\(\)](#) function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

**Note** – Remove method in List will only remove the first occurrence of the searched element.

```
# Python program to demonstrate
# Removal of elements in a Array

# importing "array" for array operations
import array

# initializing array with array values
```

```

# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 5])

# printing original array
print ("The new created array is : ", end = "")
for i in range (0, 5):
    print (arr[i], end = " ")

print ("\r")

# using pop() to remove element at 2nd position
print ("The popped element is : ", end = "")
print (arr.pop(2))

# printing array after popping
print ("The array after popping is : ", end = "")
for i in range (0, 4):
    print (arr[i], end = " ")

print ("\r")

# using remove() to remove 1st occurrence of 1
arr.remove(1)

# printing array after removing
print ("The array after removing is : ", end = "")
for i in range (0, 3):
    print (arr[i], end = " ")

```

### **Output:**

```

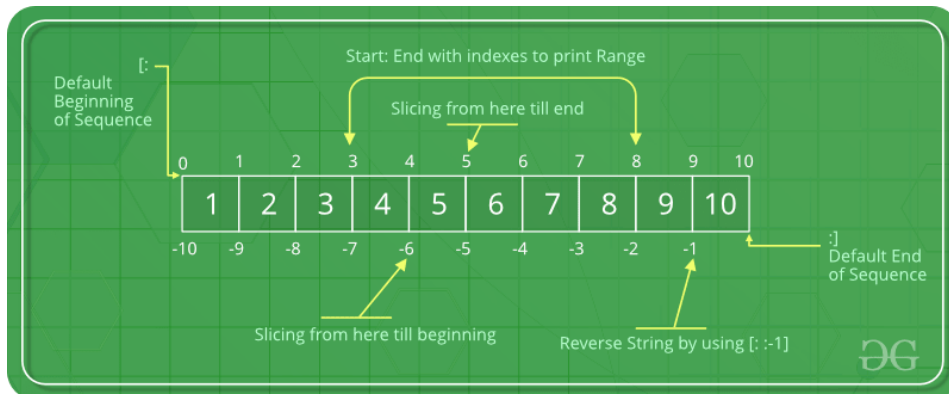
The new created array is : 1 2 3 1 5
The popped element is : 3
The array after popping is : 1 2 1 5
The array after removing is : 2 1 5

```



### Slicing of a Array

In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use [Slice operation](#). Slice operation is performed on array with the use of colon(:). To print elements from beginning to a range use [:Index], to print elements from end use[:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:]. Further, to print whole array in reverse order, use[::-1].



```
# Python program to demonstrate
# slicing of elements in a Array

# importing array module
import array as arr

# creating a list
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

a = arr.array('i', l)
print("Initial Array: ")
for i in (a):
    print(i, end = " ")

# Print elements of a range
# using Slice operation
Sliced_array = a[3:8]
```

```

print("\nSlicing elements in a range 3-8: ")
print(Sliced_array)

# Print elements from a
# pre-defined point to end
Sliced_array = a[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_array)

# Printing elements from
# beginning till end
Sliced_array = a[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_array)

```

### Output

```

Initial Array:
1 2 3 4 5 6 7 8 9 10
Slicing elements in a range 3-8:
array('i', [4, 5, 6, 7, 8])

Elements sliced from 5th element till the end:
array('i', [6, 7, 8, 9, 10])

Printing all elements using slice operation:
array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

```

### Output :

```

Initial Array:
1 2 3 4 5 6 7 8 9 10
Slicing elements in a range 3-8:
array('i', [4, 5, 6, 7, 8])

Elements sliced from 5th element till the end:
array('i', [6, 7, 8, 9, 10])

Printing all elements using slice operation:
array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

```

### *Searching element in a Array*

In order to search an element in the array we use a python in-built [index\(\)](#) method. This function returns the index of the first occurrence of value mentioned in arguments.

```
# Python code to demonstrate
# searching an element in array

# importing array module
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("The new created array is : ", end = "")
for i in range (0, 6):
    print (arr[i], end = " ")

print ("\r")

# using index() to print index of 1st occurrence of 2
print ("The index of 1st occurrence of 2 is : ", end = "")
print (arr.index(2))

# using index() to print index of 1st occurrence of 1
print ("The index of 1st occurrence of 1 is : ", end = "")
print (arr.index(1))
```

### **Output:**

```
The new created array is : 1 2 3 1 2 5
The index of 1st occurrence of 2 is : 1
The index of 1st occurrence of 1 is : 0
```

### *Updating Elements in a Array*

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

```
# Python code to demonstrate
# how to update an element in array

# importing array module
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print("Array before updation : ", end = "")
for i in range(0, 6):
    print(arr[i], end = " ")

print("\r")

# updating a element in a array
arr[2] = 6
print("Array after updation : ", end = "")
for i in range(0, 6):
    print(arr[i], end = " ")
print()

# updating a element in a array
arr[4] = 8
print("Array after updation : ", end = "")
for i in range(0, 6):
```

```
print (arr[i], end =" ")
```

**Output:**

```
Array before updation : 1 2 3 1 2 5  
Array after updation : 1 2 6 1 2 5  
Array after updation : 1 2 6 1 8 5
```