### **PYTHON KEYWORDS**

### **AND, OR , NOT -**

* AND, OR, NOT are the logical operators in Python.
* **AND** will result into True only if both the operands are True.
* **OR** will result into True if any of the operands is True.
* **NOT** operator is used to invert the truth value.

**as -**

**‘as’**is used to create an alias while importing a module. It means giving a different name (user-defined) to a module while importing it.

### **assert -**

**‘Assert’**is used for debugging purposes. While programming, sometimes we wish to know the internal state or check if our assumptions are true. ’**assert’**helps us do this and find bugs more conveniently. ’**assert’**is followed by a condition. If the condition is true, nothing happens. But if the condition is false, ”**AssertionError ”** is raised.

Syntax: assert condition, message

For Example:

>>> a = 4>>> assert a > 5, "The value of a is too small"

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

AssertionError: The value of a is too small

### **async, await -**

The ’**async’**and ’**await’**keywords are provided by the ’**asyncio library’** in Python. They are used to write concurrent code in Python.

For example:

import asyncio

async def main():

print('Hello')

await asyncio.sleep(1)

print('world')

To run the program, we use

asyncio.run(main())

In the above program,

* The **async**keyword specifies that the function will be executed asynchronously. Here, first Hello is printed.
* The **await**keyword makes the program wait for 1 second. And then the world is printed.

### **break, continue -**

* **‘break’**and ’**continue’**are used inside **for**and **while loops** to alter their normal behavior.
* **‘break’**will end the smallest loop it is in and control flows to the statement immediately below the loop break out of loop and passes the control to the statement following immediately after loop.
* **‘continue’**causes to end the current iteration of the loop, but not the whole loop. The keyword skips the current iteration of the loop, but does not end the loop.

For Example:

for i in range(1,8):

if i == 5:

break

print(i)

**OUTPUT:**

1

2

3

4

for i in range(1,8):

if i == 5:

continue

print(i)

**OUTPUT:**

1

2

3

4

6

7

8

### **class -**

* **‘Class’**is used to define a new user-defined class in Python.
* **‘Class’** is a collection of related attributes and methods that try to represent a real world situation. This idea of putting data and functions together in a class is central to the concept of object-oriented programming (OOP).
* **Classes** can be defined anywhere in a program. But it is a good practice to define a single class in a module.

SYNTAX class ExampleClass:

def function1(parameters):

…

def function2(parameters):

…

### **def -**

* **‘def ‘**is used to define a user-defined function.
* Function is a block of related statements, which together does some specific task. It helps us organize code into manageable chunks and also to do some repetitive task.

SYNTAX: def function\_name(parameters):

…

### **del -**

* **‘del’**is used to delete the reference to an object. Everything is object in Python.
* We can delete a variable reference using **del.**
* **‘del’**is also used to delete items from a list or a dictionary.

For Example:

>>> a = b = 5

>>> del a

>>> a

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

NameError: name 'a' is not defined

>>> b

5

Here we can see that the reference of the variable a was deleted. So, it is no longer defined. But b still exists.

### **if, else, elif -**

* **‘if ‘** , ‘**else’** , ‘**elif ‘**are used for conditional branching or decision making.
* When we want to test some condition and execute a block only ifthe condition is **true**, then we use **if**and **elif**. **‘elif ‘**is short for else if.
* ‘**else’**is the block which is executed if the condition is false.

For Example:

def if\_example(a):

if a == 1:

print('One')

elif a == 2:

print('Two')

else:

print('Something else')

if\_example(2)

if\_example(4)

### **except, raise, try -**

* **‘except’** , **‘raise’** , **‘try’**are used with exceptions in Python.
* **Exceptions** are basically errors that suggests something went wrong while executing our program.
* IOError, ValueError, ZeroDivisionError, ImportError, NameError, TypeError etc. are few examples of exception in Python.
* **try...except blocks** are used to catch exceptions in Python.
* We can raise an exception explicitly with the **’raise’** keyword.

def reciprocal(num):

try:

r = 1/num

except:

print('Exception caught')

return

return r

print(reciprocal(10))

print(reciprocal(0))

**OUTPUT:**

0.1

Exception caught

None

Here, the function **reciprocal()** returns the reciprocal of the input number.When we enter 10, we get the normal output of 0.1. But when we input 0, a**ZeroDivisionError** is raised automatically. This is caught by our **try…except block** and we return None.

We could have also raised the **ZeroDivisionError**explicitly by checking the input and handled it elsewhere as follows:

if num == 0:

raise ZeroDivisionError('cannot divide')

### **finally -**

* **finally**is used with **try…except block** to close up resources or file streams.
* Using **finally**ensures that the block of code inside it gets executed even if there is an unhandled exception.

For example:

try:

Try-blockexcept exception1:

Exception1-blockexcept exception2:

Exception2-blockelse:

Else-blockfinally:

Finally-block

Here if there is an exception in the **Try-block**, it is handled in the **except**or **else**block. But no matter in what order the execution flows, we can rest assured that the**Finally-block** is executed even if there is an error. This is useful in cleaning up the resources.

### **for -**

* **‘for’** is used for looping. Generally we use for when we know the number of times we want to loop.
* In Python we can use it with any type of sequence like a list or a string.

For Example:

names = ['John','Monica','Steven','Robin']for i in names:

print('Hello '+i)

### **while -**

* ‘**while’**is used for looping in Python.
* The statements inside a while loop continue to execute until the condition for the **while loop** evaluates to **False**or a **break**statement is encountered.

For Example:

i = 5

while(i):

print(i)

i = i – 1

### **from, import -**

* **‘import’**keyword is used to import modules into the current namespace.
* **‘from…import’** is used to import specific attributes or functions into the current namespace.

For example:

import math

will import the math module. Now we can use the cos() function inside it as math.cos(). But if we wanted to import just the cos() function, this can done using from as

from math import cos

now we can use the function simply as cos(), no need to write math.cos().

### **global -**

* **‘global’**is used to declare that a variable inside the function is global (outside the function).
* If we need to read the value of a **global** variable, it is not necessary to define it as global. This is understood.
* If we need to modify the value of a **global** variable inside a function, then we must declare it with global. Otherwise a local variable with that name is created.

For Example:

globvar = 10def read1():

print(globvar)def write1():

global globvar

globvar = 5def write2():

globvar = 15

read1()

write1()

read1()

write2()

read1()

**OUTPUT:**

10

5

5

We can see in our output that the modification did take place (10 is changed to 5). The **write2()** also tries to modify this value. But we have not declared it as **global**.

### **in -**

* **‘in’** is used to test if a sequence (list, tuple, string etc.) contains a value. It returns **True**if the value is present, else it returns **False**.

For example:

>>> a = [1, 2, 3, 4, 5]

>>> 5 in a

True

>>> 10 in a

False

* The secondary use of **in**is to traverse through a sequence in a **for**loop.

for i in 'hello':

print(i)

### **is -**

* **‘is’**is used in Python for testing **object** identity. While the **==**operator is used to test if two variables are **equal or not**,
* **‘is’** is used to test if the two variables refer to the same object.
* It returns **True**if the objects are identical and **False**if not.

>>> [] == []True

>>> [] is []False

>>> {} == {}True

>>> {} is {}False

* An empty list or dictionary is equal to another empty one. But they are not identical objects as they are located separately in memory. **This is because list and dictionary are mutable** (value can be changed).
* **string** and **tuple are immutable** (value cannot be altered once defined). Hence, two equal string or tuple are identical as well. They refer to the same memory location.

### **lambda -**

* **“lambda”**is used to create an anonymous function (function with no name).
* It is an inline function that does not contain a **return**statement. It consists of an expression that is evaluated and returned.

For example:

a = lambda x: x\*2for i in range(1,6):

print(a(i))

**OUTPUT:**

2

4

6

8

10

### **nonlocal -**

* The use of ”**nonlocal”**keyword is very much similar to the global keyword.
* “**nonlocal”**is used to declare that a variable inside a nested function (function inside a function) is not local to it, meaning it lies in the outer inclosing function.
* If we need to modify the value of a non-local variable inside a nested function, then we must declare it with nonlocal. Otherwise a local variable with that name is created inside the nested function.

For Example:

def outer\_function():

a = 5

def inner\_function():

nonlocal a

a = 10

print("Inner function: ",a)

inner\_function()

print("Outer function: ",a)

outer\_function()

**OUTPUT:**

Inner function: 10

Outer function: 10

### **pass -**

* **“pass”**is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.

def function(args):

pass

* We can do the same thing in an **empty class** as well.

class example:

pass

### **with -**

* **“with”** statement is used to wrap the execution of a block of code within methods defined by the context manager.
* Context manager is a class that implements **\_\_enter\_\_**and **\_\_exit\_\_**methods.
* Use of **with**statement ensures that the **\_\_exit\_\_**method is called at the end of the nested block. This concept is similar to the use of **try…finally** block.

For Example:

with open('example.txt', 'w') as my\_file:

my\_file.write('Hello world!')

* File objects have**\_\_enter\_\_** and**\_\_exit\_\_** method defined within them, so they act as their own context manager.
* First the **\_\_enter\_\_**method is called, then the code within **with**statement is executed and finally the **\_\_exit\_\_** method is called.
* **\_\_exit\_\_**method is called even if there is an error. It basically closes the file stream.

### **yield -**

* **“yield”**is used inside a function like a **return**statement. But **yield**returns a generator.
* Generator is an iterator that generates one item at a time. A large list of value will take up a lot of memory. Generators are useful in this situation as it generates only one value at a time instead of storing all the values in memory.

For example:

>>> g = (2\*\*x for x in range(100))

will create a generator, **g**which generates powers of 2 up to the number two raised to the power 99. We can generate the numbers using the **next()**function as shown below.

>>> next(g)

1

>>> next(g)

2

>>> next(g)

4

>>> next(g)

8

And so on… This type of generator is returned by the **yield**statement from a function. Here is an example.

def generator():

for i in range(6):

yield i\*I

g = generator()for i in g:

print(i)

**OUTPUT:**

0 1 4 9 16 2

**PYTHON BASIC**

**Input/Output**

* *Taking multiple inputs from user in Python:*

Python user can take multiple values or inputs in one line by two methods--

* Using split( ) method : This function helps in getting a multiple inputs from user . It breaks the given input by the specified separator.

**SYNTAX :** input( ).split(separator, maxsplit)

**EXAMPLE**:

# taking two inputs at a time

a, b **=** input("Enter a two value: ").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 a multiple value: ").split()))

print("List of students: ", x)

* 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.

**EXAMPLE**:

# taking multiple inputs at a time

x **=** [int(x) **for** x **in** input("Enter multiple value: ").split()]

**print**("Number of list is: ", x)

* *Input Methods for Competitive Programming:*

A bit faster method using ***inbuilt -* stdin**, **stdout** :  
1.) **sys.stdin** on the other hand is a File Object. It is like creating any other file object one could create to read input from the file. In this case, the file will be standard input buffer.  
2.) **stdout.write(‘D\n’)** is faster than **print** ‘D’.  
3.) Even faster is to write all once by **stdout.write**(“”.join(list-comprehension)) but this makes memory usage dependent on size of input.

**EXAMPLE**:

# import inbuilt standard input output

**from** sys **import** stdin, stdout

# suppose a function called main() and

# all the operations are performed

**def** main():

    # input via readline method

    n **=** stdin.readline()

    # array input similar method

    arr **=** [int(x) **for** x **in** stdin.readline().split()]

    #initialize variable

    summation **=** 0

    # calculate sum

**for** x **in** arr:

        summation **+=** x

    # could use inbuilt summation = sum(arr)

    # print answer via write, write method writes only string operations

    # so we need to convert any data into string for input

    stdout.write(str(summation))

# call the main method

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    main()

* *Output using print( ) function:*

Using the print() function where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string before writing to the screen --

* ***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.*

**EXAMPLE:**

# 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")

* *Python | end parameter in print():*

By default python’s print() function ends with a newline.

Python’s print( ) function comes with a parameter called ‘end’. By default, the value of this parameter is ‘\n’, i.e. the new line character. You can end a print statement with any character/string using this parameter.

**EXAMPLE:**

# array

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

**# printing an element in same line to do so we use *end = “sep” inside print( )***

**for** i **in** range(4):

print(a[i], end **=**" ")

**OUTPUT:**

1 2 3 4 5

* *Python | sep parameter in print():*

The separator between the arguments to print() function in Python is space by default (softspace feature) , which can be modified and can be made to any character, integer or string as per our choice.

**EXAMPLE:**

#for formatting a date

print('09','12','2016', sep**=**'-')

#another example

print('pratik','geeksforgeeks', sep**=**'@')

**OUTPUT:**

09-12-2016

pratik@geeksforgeeks

* *Python | Output formatting:*

Format method of strings requires more manual effort. User use {} to mark where a variable will be substituted and can provide detailed formatting directives, but user also needs to provide the information to be formatted.

**EXAMPLE:**

# print integer and float value

**print**("Geeks : % 2d, Portal : % 5.2f" **%**(1, 05.333))

# using format() method

**print**('I love {} for "{}!"'.format('Geeks', 'Geeks'))

# using format() method and refering

# a position of the object

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

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

**OUTPUT:**

Geeks : 1, Portal : 5.33

I love Geeks for “Geeks!”

Geeks and Portal

Portal and Geeks

**Data Type**

* *Python | String:*

Strings are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets [ ] can be used to access elements of the string.

* **Creating a String:** Strings in Python can be created using single quotes or double quotes or even triple quotes
* **Accessing characters in Python:** In Python, individual characters of a String can be accessed by using the method of Indexing.While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types will cause a **TypeError**. Also negative index allows that; -1 refers to the last character, -2 refers to the second last character and so on.
* **String Slicing:** To access a range of characters in the String, method of **slicing** is used. Slicing in a String is done by using a Slicing operator (colon).

String1 **=** "GeeksForGeeks"

# Printing 3rd to 12th character

print("\nSlicing characters from 3-12: ")

**print**(String1[3:12])

**OUTPUT:**

Slicing characters from 3-12:

ksForGeek

* **Deleting/Updating from a String:** In Python, Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name.
* **Escape Sequencing in Python:** While printing Strings with single and double quotes in it then it causes SyntaxError because String already contains Single and Double Quotes. Hence, to print such a String either *Triple Quotes or* *Escape sequences start with a backslash* can be used to print such Strings.

# Initial String

String1 **=** '''I'm a "Geek"'''

**print**("Initial String with use of Triple Quotes: ", end = “”)

print(String1)

# Escaping Doule Quotes

String1 **=** "I'm a \"Geek\""

**print**("\nEscaping Double Quotes: ", end = “”)

print(String1)

**OUTPUT:**

Initial String with use of Triple Quotes: I'm a "Geek"

Escaping Double Quotes: I'm a "Geek"

* *Python | List:*

Strings are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets [ ] can be used to access elements of the string.

* **Creating a List:** Lists in Python can be created by just placing the sequence inside the square brackets[ ]. **Note –** Unlike Sets, list may contain mutable elements.
* **Knowing Size of List:** List uses *len( )* function to know the length of list, string, tuple and dictionary object.

# Creating a List of numbers

List2 **=** [10, 20, 14]

print(len(List2))

**OUTPUT:**

3

* **Adding Elements to a List:**
  + Append( ) method -- Elements can be added to the List by using built-in *[append( )](https://www.geeksforgeeks.org/list-methods-python/" \t "https://www.geeksforgeeks.org/python-list/_blank)* function. Only one element at a time can be added to the list by using *append( )* method, for addition of multiple elements with the append( ) method, loops are used.
  + Insert( ) method -- As *append( )* method only works for addition of elements at the end of the List, for addition of element at the desired position,*insert( )*method is used. Unlike *append( )*which takes only one argument, *insert( )* method requires two arguments*[position, value]*.
  + Extend( ) method -- Other than *append( )* and *insert( )* methods, there’s one more method for Addition of elements, *[extend( )](https://www.geeksforgeeks.org/append-extend-python/" \t "https://www.geeksforgeeks.org/python-list/_blank)*, this method is used to add multiple elements at the same time at the end of the list. **Note –** *[append( )](https://www.geeksforgeeks.org/append-extend-python/" \t "https://www.geeksforgeeks.org/python-list/_blank)* [and](https://www.geeksforgeeks.org/append-extend-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) *[extend( )](https://www.geeksforgeeks.org/append-extend-python/" \t "https://www.geeksforgeeks.org/python-list/_blank)* methods can only add elements at the end.

# Creating a List

List **=** []

# Adding elements to the List using Iterator

**for** i **in** range(1, 4):

    List.append(i)

**print**(List)

# Addition of Element at specific Position(using Insert Method)

List.insert(3, 12)

List.insert(0, 'Geeks')

**print**(List)

# Addition of multiple elements to the List at the end(using Extend Method)

List.extend([8, 'Geeks', 'Always'])

**print**(List)

**OUTPUT:**

[1, 2, 3, 4]

['Geeks', 1, 2, 3, 12, 4]

[1, 2, 3, 4, 8, 'Geeks', 'Always']

* **Removing Elements from the List:**
  + Remove( ) method -- Elements can be removed from the List by using built-in*[remove( )](https://www.geeksforgeeks.org/python-list-remove/" \t "https://www.geeksforgeeks.org/python-list/_blank)* function but an Error arises if element doesn’t exist in the set. *[Remove( )](https://www.geeksforgeeks.org/python-list-remove/" \t "https://www.geeksforgeeks.org/python-list/_blank)* method only removes one element at a time, to remove range of elements, iterator is used. The *remove( )* method removes the specified item. **Note –** Remove method in List will only remove the first occurrence of the searched element.
  + Pop( ) method -- *[Pop( )](https://www.geeksforgeeks.org/python-list-pop/" \t "https://www.geeksforgeeks.org/python-list/_blank)* function can also be used to remove and return an element from the set, but by default it removes only the last element of the set, to remove element from a specific position of the List, index of the element is passed as an argument to the *pop( )* method.

# Creating a List

List **=** [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# Removing elements from List using iterator method(by remove())

**for** i **in** range(1, 5):

    List.remove(i)

**print**(List)

# Removing element at a specific location from the Set using the pop() method

List.pop(2)

**print**(List)

**OUTPUT:**

[7, 8, 9, 10, 11, 12]

[7, 8, 10, 11, 12]

* **Slicing of a List:** In Python List, there are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use [Slice operation](https://www.geeksforgeeks.org/python-list-comprehension-and-slicing/" \t "https://www.geeksforgeeks.org/python-list/_blank). Slice operation is performed on Lists with the use of colon. **Note –** To print elements of List from rear end, use Negative Indexes. **SYNTAX:**- *Lists[start\_index:end\_index]*
* **List Methods:**

|  |  |
| --- | --- |
| [Clear( )](https://www.geeksforgeeks.org/list-methods-in-python-set-2-del-remove-sort-insert-pop-extend/) | Removes all items from the list |
| [Index( )](https://www.geeksforgeeks.org/python-list-index/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Returns the index of the first matched item | **SYNTAX:** list\_name.index(element, start, end) |
| [Count( )](https://www.geeksforgeeks.org/python-list-function-count/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Returns the count of number of items passed as an argument | **SYNTAX:** list\_name.count(object) |
| [Sort( )](https://www.geeksforgeeks.org/sort-in-python/) | Sort items in a list in ascending order | **SYNTAX:** List\_name.sort( ) ; List\_name.sort(reverse=True) |
| [Reverse( )](https://www.geeksforgeeks.org/list-methods-in-python-set-2-del-remove-sort-insert-pop-extend/) | Reverse the order of items in the list | **SYNTAX:** List\_name.reverse( ) |
| [copy( )](https://www.geeksforgeeks.org/python-list-copy-method/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Returns a copy of the list | **SYNTAX:** List\_name.copy( ) **or** lis3 **=** copy.deepcopy(lis1) |

**Built-in functions with List**

|  |  |
| --- | --- |
| FUNCTION | DESCRIPTION |
| [reduce()](https://www.geeksforgeeks.org/reduce-in-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | apply a particular function passed in its argument to all of the list elements stores the intermediate result and only returns the final summation value |
| [sum()](https://www.geeksforgeeks.org/sum-function-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Sums up the numbers in the list |
| [ord()](https://www.geeksforgeeks.org/ord-function-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Returns an integer representing the Unicode code point of the given Unicode character |
| [cmp()](https://www.geeksforgeeks.org/python-2-number-cmplist-method/" \t "https://www.geeksforgeeks.org/python-list/_blank) | This function returns 1, if first list is “greater” than second list |
| max() | return maximum element of given list |
| min() | return minimum element of given list |
| [all()](https://www.geeksforgeeks.org/any-all-in-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Returns true if all element are true or if list is empty |
| [any()](https://www.geeksforgeeks.org/any-all-in-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | return true if any element of the list is true. if list is empty, return false |
| len() | Returns length of the list or size of the list |
| [enumerate()](https://www.geeksforgeeks.org/enumerate-in-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | Returns enumerate object of list |
| accumulate() | apply a particular function passed in its argument to all of the list elements returns a list containing the intermediate results |
| [filter()](https://www.geeksforgeeks.org/filter-in-python/" \t "https://www.geeksforgeeks.org/python-list/_blank) | tests if each element of a list true or not |
| [map()](https://www.geeksforgeeks.org/python-map-function/" \t "https://www.geeksforgeeks.org/python-list/_blank) | returns a list of the results after applying the given function to each item of a given iterable |
| [lambda()](https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/" \t "https://www.geeksforgeeks.org/python-list/_blank) | This function can have any number of arguments but only one expression, which is evaluated and returned. |

* *Python | Tuple:*

**Tuple**is a collection of Python objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers. Tuple is closing the sequence of values in parentheses( ) and immutable.

* **Concatenation of Tuples:** *Concatenation* of tuple is the process of joining of 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 same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.*

# Concatenaton of tuples

Tuple1 **=** (0, 1, 2, 3)

Tuple2 **=** ('Geeks', 'For', 'Geeks')

Tuple3 **=** Tuple1 **+** Tuple2

print(Tuple3)

**OUTPUT:**

(0, 1, 2, 3, 'Geeks', 'For', 'Geeks')

* **Slicing of Tuple , Deleting of Tuple as same as *List:***
* **Enumerate of Tuple:** *Enumerate( ) method* adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in for loops or be converted into a list of tuples using list( ) method.

**SYNTAX:** enumerate(iterable, start=0)

# Python program to illustrate enumerate function

l1 **=** ["eat","sleep","repeat"]

# creating enumerate objects

obj1 **=** enumerate(l1)

**print** list(enumerate(l1))

# changing start index to 2 from 0

print list(enumerate(l1,2))

**OUTPUT:**

[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]

[(2, 'eat'), (3 'sleep'), (4, 'repeat')]

* **Built-In Methods:**

|  |  |
| --- | --- |
| BUILT-IN FUNCTION | DESCRIPTION |
| all() | Returns true if all element are true or if tuple is empty |
| any() | return true if any element of the tuple is true. if tuple is empty, return false |
| len() | Returns length of the tuple or size of the tuple |
| enumerate() | Returns enumerate object of tuple |
| max() | return maximum element of given tuple |
| min() | return minimum element of given tuple |
| [sum()](https://www.geeksforgeeks.org/sum-function-python/" \t "https://www.geeksforgeeks.org/python-tuples/_blank) | Sums up the numbers in the tuple |
| [sorted()](https://www.geeksforgeeks.org/sorted-function-python/" \t "https://www.geeksforgeeks.org/python-tuples/_blank) | input elements in the tuple and return a new sorted list |
| [tuple()](https://www.geeksforgeeks.org/python-tuple-function/" \t "https://www.geeksforgeeks.org/python-tuples/_blank) | Convert an iterable to a tuple. |

* *Python | Set:*

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

The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

**Note –**A set cannot have mutable elements like a list, set or dictionary, as its elements.

* **Creating a Set:** Sets can be created by using the built-in **set( )** **function** with an *iterable object* or *a sequence* by placing the sequence inside curly braces, separated by ‘comma’.

# Creating a Set with a List of Numbers(Having duplicate values)

set1 **=** set([1, 2, 4, 4, 3, 3, 3, 6, 5])

**print**(set1)

# Creating a Set with mixed type of values(Having numbers and strings)

set1 **=** set([1, 2, 'Geeks', 4, 'For', 6, 4, 'Geeks'])

**print**(set1)

**OUTPUT:**

{1, 2, 3, 4, 5, 6}

{1, 2, 4, 'Geeks', 6, 'For'}

* **Adding Elements to a Set:**
* Add( ) method -- Elements can be added to the Set by using built-in **add( )** function. Only one element at a time can be added to the set by using add( ) method, loops are used to add multiple elements at a time with the use of add( ) method.

***Note –****Lists cannot be added to a set as elements because Lists are not hashable whereas Tuples can be added because tuples are immutable and hence Hashable.*

* Update() method -- For addition of two or more elements **Update()**method is used. The update() method accepts lists, strings, tuples as well as other sets as its arguments.

***Note –****In all of these cases, duplicate elements are avoided.*

# Addition of elements to the Set using Update function

set1 **=** set([ 4, 5, (6, 7)])

set1.update([10, 11])

print(set1)

**OUTPUT:**

{10, 11, 4, 5, (6, 7)}

* **Removing elements from the Set:**
* 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.
* Pop() method -- Pop() function can also be used to remove and return an element from the set, but it removes only the last element of the set.

***Note –****If the set is unordered then there’s no such way to determine which element is popped by using the pop( ) function.*

* Clear() method -- To remove all the elements from the set, clear() function is used.

# Creating a Set

set1 **=** set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

# Removing elements from Set using Remove() method

set1.remove(5)

**print**(set1)

# Removing elements from Set using Discard() method

set1.discard(8)

**print**(set1)

# Removing element from the Set using the pop() method

set1.pop()

**print**(set1)

# Removing all the elements from Set using clear() method

set1.clear()

**print**(set1)

**OUTPUT:**

{1, 2, 3, 4, 7, 8, 9, 10, 11, 12}

{1, 2, 3, 4, 7, 10, 11, 12}

{ 2, 3, 4, 7, 10, 11,12}

set( )

* *Python | Dict:*

In Python, Dictionary is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds ***key:value*** pair.

Dictionary can also be created by the built-in function **dict( )**. An empty dictionary can be created by just placing to curly braces{ }.

* **Creating a Dictionary:** In Python, a Dictionary can be created by placing sequence of elements within curly { } braces, separated by ‘comma’. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its ***Key:value***. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can’t be repeated and must be immutable.

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

# Creating a Dictionary with Mixed keys

Dict **=** {'Name': 'Geeks', 1: [1, 2, 3, 4]}

**print**(Dict)

# Creating a Dictionary with dict() method

Dict **=** dict({1: 'Geeks', 2: 'For', 3:'Geeks'})

**print**(Dict)

# Creating a Nested Dictionary as shown in the below image

Dict **=** {1: 'Geeks', 2: 'For',

        3:{'A' : 'Welcome', 'B' : 'To', 'C' : 'Geeks'}}

**print**(Dict)

**OUTPUT:**

{1: [1, 2, 3, 4], 'Name': 'Geeks'}

{1: 'Geeks', 2: 'For', 3: 'Geeks'}

{1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}

* **Adding elements to a Dictionary:** Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. ***Dict[Key] = ‘Value’***.  
  *Note- While adding a value, if the key value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.*
* **Accessing elements from a Dictionary:** In order to access the items of a dictionary refer to its **key** name. **Key** can be used inside square brackets[ ]. There is also a method called **[get( )](https://www.geeksforgeeks.org/get-method-dictionaries-python/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank)** that will also help in acessing the element from a dictionary.

# Creating a Dictionary

Dict **=** {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key and get() method

**print**(Dict['name'])

print(Dict.get(3))

**OUTPUT:**

For

Geeks

# Creating a Dictionary

Dict **=** {'Dict1': {1: 'Geeks'},

        'Dict2': {'Name': 'For'}}

# Accessing element using key

**print**(Dict['Dict1'])

**print**(Dict['Dict1'][1])

print(Dict['Dict2']['Name'])

**OUTPUT:**

{1: 'Geeks'}

Geeks

For

* **Removing Elements from Dictionary:** Using *del( )*, *pop( )*, *popitem( )*, *clear( )* methods we can remove element of specified keys in dictionary.
* **Dictionary Methods:**

|  |  |
| --- | --- |
| METHODS | DESCRIPTION |
| [copy()](https://www.geeksforgeeks.org/python-dictionary-copy/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | They copy() method returns a shallow copy of the dictionary. |
| [clear()](https://www.geeksforgeeks.org/python-dictionary-clear/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | The clear() method removes all items from the dictionary. |
| [pop()](https://www.geeksforgeeks.org/python-dictionary-pop-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Removes and returns an element from a dictionary having the given key. |
| [popitem()](https://www.geeksforgeeks.org/python-dictionary-popitem-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Removes the arbitrary key-value pair from the dictionary and returns it as tuple. |
| [get()](https://www.geeksforgeeks.org/get-method-dictionaries-python/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | It is a conventional method to access a value for a key. |
| [dictionary\_name.values()](https://www.geeksforgeeks.org/python-dictionary-values/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | returns a list of all the values available in a given dictionary. |
| str() | Produces a printable string representation of a dictionary. |
| [update()](https://www.geeksforgeeks.org/python-dictionary-update-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Adds dictionary dict2’s key-values pairs to dict |
| [setdefault()](https://www.geeksforgeeks.org/python-dictionary-setdefault-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Set dict[key]=default if key is not already in dict |
| [keys()](https://www.geeksforgeeks.org/python-dictionary-keys-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Returns list of dictionary dict’s keys | **Syntax:** dictionary.keys() |
| [items()](https://www.geeksforgeeks.org/python-dictionary-items-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Returns a list of dict’s (key, value) tuple pairs | **Syntax:** dictionary.items() |
| [has\_key()](https://www.geeksforgeeks.org/python-dictionary-has_key/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Returns true if key in dictionary dict, false otherwise |
| [fromkeys()](https://www.geeksforgeeks.org/python-dictionary-fromkeys-method/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Create a new dictionary with keys from seq and values set to value. | **Syntax:** dict.fromkeys(seq, 1) |
| [type()](https://www.geeksforgeeks.org/python-type-function/" \t "https://www.geeksforgeeks.org/python-dictionary/_blank) | Returns the type of the passed variable. |
| [cmp()](https://www.geeksforgeeks.org/dictionary-methods-in-python-set-1-cmp-len-items/) | Compares elements of both dict. |

* *Python | Array:*

Array can be handled in python by module named **“array“.** They can be useful when we have to manipulate only a specific data type values.

**Operations on Array :**

* array(data type, value list) :- This function is used to create an array with data type and value list specified in its arguments.
* append( ) and insert(i,x)  :- This function is used to add the value mentioned in its arguments at the end of the array and insert(i,x) function is used to add the value at the position specified in its argument.
* pop( ) and remove( ) :- pop( ) function removes the element at the position mentioned in its argument, and returns it and remove( ) function is used to remove the first occurrence of the value mentioned in its arguments.
* count( ) :- This function counts the number of occurrences of argument mentioned in array.
* extend(arr) :- This function appends a whole array mentioned in its arguments to the specified array.
* fromlist(list) :- This function is used to append a list mentioned in its argument to end of array.
* tolist( ) :- This function is used to transform an array into a list.

# 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, 2, 5])

# initializing list

li **=** [1, 2, 3]

# using fromlist() to append list at end of array

arr.fromlist(li)

**for** i **in** range (0,9):

**print** (arr[i],end**=**" ")

# using tolist() to convert array into list

li2 **=** arr.tolist()

**for** i **in** range (0,len(li2)):

print (li2[i],end**=**" ")

# using count() to count occurrences of 1 in array

**print** (arr1.count(1))

**OUTPUT:**

The modified array is : 1 2 3 1 2 5 1 2 3

The new list created is : 1 2 3 1 2 5 1 2 3

3

**Variables**

* *Global and Local Variables:*

Global variables are the one that are defined and declared outside a function and we need to use them inside a function. To tell Python, that we want to use the global variable, we have to use the keyword **“global”**. If a variable with the same name is defined inside the scope of function is **“local”** variable as well then it will print the value given inside the function only and not the global value.

# This function modifies the global variable 's'

**def** f():

**global** s

    print s

    s **=** "Look for Geeksforgeeks Python Section"

**print** s

# Global Scope

s **=** "Python is great!"

f()

**print** s

**OUTPUT:**

Python is great!

Look for Geeksforgeeks Python Section.

Look for Geeksforgeeks Python Section.

* *Packing and Unpacking Arguments:*

We use two operators **\* (for tuples)** and **\*\* (for dictionaries)** for Packing and Unpacking arguments. When we call for a function by passing the list( ) as an argument for the function( ) parameters then that call doesn’t work and it shows ERROR.

* **Unpacking:** We can use \* to unpack the list so that all elements of it can be passed as different parameters.

# A sample function that takes 4 arguments

**def** fun(a, b, c, d):

**print**(a, b, c, d)

# Driver Code

my\_list **=** [1, 2, 3, 4]

# Unpacking list into four arguments

fun(**\***my\_list)

**OUTPUT:**

(1, 2, 3, 4)

* **Packing:** When we don’t know how many arguments need to be passed to a python function, then we can use packing to pack all arguments in a tuple.

# This function uses packing to sum unknown number of arguments

**def** mySum(**\***args):

    sum **=** 0

**for** i **in** range(0, len(args)):

        sum **=** sum **+** args[i]

**return** sum

# Driver code

**print**(mySum(1, 2, 3, 4, 5))

**print**(mySum(10, 20))

**OUTPUT:**

15

30

* Here **\*\*** unpacked the dictionary used with it, and passed the items in the dictionary as keyword arguments to the function. So writing “fun(1, \*\*d)” was equivalent to writing “fun(1, b=4, c=10)”.

# A sample program to demonstrate unpacking of

# dictionary items using \*\*

**def** fun(a, b, c):

    print(a, b, c)

# A call with unpacking of dictionary

d **=** {'a':2, 'b':4, 'c':10}

fun(**\*\***d)

**OUTPUT:**

2 4 10

# A Python program to demonstrate packing of dictionary items using \*\*

**def** fun(**\*\***kwargs):

    # kwargs is a dict

**print**(type(kwargs))

    # Printing dictionary items

**for** key **in** kwargs:

**print**("%s = %s" **%** (key, kwargs[key]))

# Driver code

fun(name**=**"geeks", ID**=**"101", language**=**"Python")

**OUTPUT:**

<class 'dict'>

language = Python

name = geeks

ID = 101

*Applications and Important Points:-  
1. Used in socket programming to send a vast number of requests to a server.  
2. Used in Django framework to send variable arguments to view functions.  
3. There are wrapper functions that require us to pass in variable arguments.  
4. Modification of arguments become easy, but at the same time validation is not proper, so they must be used with care.*

* *Type Conversions:*
* **chr(number)** **:** This function converts **number** to its corresponding **ASCII** character.
* **dict( ) :** This function is used to convert a **tuple** of **order (key,value)** into a **dictionary**.
* **set( ) :** This function returns the **type** after **converting to set**.

s **=** 'geeks'

# printing string converting to tuple

**print** (tuple(s))

# printing string converting to set

**print** (set(s))

# printing tuple converting to expression dictionary

tup **=** (('a', 1) ,('f', 2), ('g', 3))

**print** (dict(tup))

# Convert ASCII value to characters

**print**(chr(76), chr(77))

**OUTPUT:**

('g', 'e', 'e', 'k', 's')

{'k', 'e', 's', 'g'}

{'a': 1, 'f': 2, 'g': 3}

L M

***Zip() in Python:*** The purpose of zip( ) is to map the similar index of multiple containers so that they can be used just using as single entity. **SYNTAX:** zip(\*iterators) **EXAMPLE: >>> name** = [ "Manjeet", "Nikhil", "Shambhavi"]; **marks** = [ 40, 50, 60] **>>>** zip(name, marks) **OUTPUT:** [('Manjeet', 40), ('Nikhil', 50), ('Shambhavi', 60)]

* *Byte vs String Objects:*

Byte objects are sequence of Bytes, whereas Strings are sequence of characters. There are methods to convert a byte object to String and String to byte objects.

* **Encode( )** **:** Converting **Strings** to **byte objects** is termed as *encoding*. This is necessary so that the text can be stored on disk using mapping using **ASCII**or **UTF-8** encoding techniques.
* **Decode( ) :** *Decoding* is process to convert a **Byte object** to **String**. It is implemented using **decode( )** . A byte string can be decoded back into a character string.

# initialising a String

a **=** 'GeeksforGeeks'

# initialising a byte object

c **=** b'GeeksforGeeks'

# using encode() to encode the String encoded version of a is stored in d using ASCII mapping

d **=** a.encode('ASCII')

# checking if a is converted to bytes or not

**if** (d**==**c):

    print ("Encoding successful")

**else** : print ("Encoding Unsuccessful")

# using decode() to decode the Byte object decoded version of c is stored in d using ASCII mapping

d **=** c.decode('ASCII')

# checking if c is converted to String or not

**if** (d**==**a):

    print ("Decoding successful")

**else** : print ("Decoding Unsuccessful")

* *Private Variables in Python:*

In Python, there is something called name mangling, which means that there is a limited support for a valid use-case for class-private members basically to avoid name clashes of names with names defined by subclasses.

Any identifier of the form \_\_geek (at least two leading underscores or at most one trailing underscore) is replaced with \_classname\_\_geek, where classname is the current class name with leading underscore(s) stripped. As long as it occurs within the definition of the class, this mangling is done.

# Python code to illustrate how mangling works

**class** Map:

**def** \_\_init\_\_(self, iterate):

        self.list **=** []

        self.\_\_geek(iterate)

**def** geek(self, iterate):

**for** item **in** iterate:

            self.list.append(item)

    # private copy of original geek() method

    \_\_geek **=** geek

**class** MapSubclass(Map):

    # provides new signature for geek() but

    # does not break \_\_init\_\_()

**def** geek(self, key, value):

**for** i **in** zip(keys, values):

            self.list.append(i)

* **\_\_Double Leading Underscores:** Two underlines, in the beginning, cause a lot of confusion. This is about syntax rather than a convention. double underscore will mangle the attribute names of a class to avoid conflicts of attribute names between classes.

# Python code to illustrate how double

# underscore at the beginning works

**class** Geek:

**def** \_single\_method(self):

**pass**

**def** \_\_double\_method(self): # for mangling

**pass**

**class** Pyth(Geek):

**def** \_\_double\_method(self): # for mangling

**pass**

* **\_\_Double leading and Double trailing underscores\_\_:** There’s another case of double leading and trailing underscores. We follow this while using special variables or methods (called “magic method”) such as\_\_len\_\_, \_\_init\_\_. These methods provide special syntactic features to the names. For example, \_\_file\_\_ indicates the location of Python file, \_\_eq\_\_ is executed when a == b expression is executed.

# Python code to illustrate double leading and

# double trailing underscore works

**class** Geek:

    # '\_\_init\_\_' for initializing, this is a

    # special method

**def** \_\_init\_\_(self, ab):

        self.ab **=** ab

    # custom special method. try not to use it

**def** \_\_custom\_\_(self):

**pass**

* **\_\_name\_\_ (A Special variable):** Since there is no main() function in Python, when the command to run a python program is given to the interpreter, the code that is at level 0 indentation is to be executed. However, before doing that, it will define a few special variables. *If the source file is executed as the main program, the interpreter sets the \_\_name\_\_ variable to have a value “\_\_main\_\_”. If this file is being imported from another module, \_\_name\_\_ will be set to the module’s name.*

**\_\_name\_\_ is a built-in variable which evaluates to the name of the current module.**Thus it can be used to check whether the current script is being run on its own or being imported somewhere else by combining it with if statement,

# File1.py

print "File1 \_\_name\_\_ = %s" **%**\_\_name\_\_

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    print "File1 is being run directly"

**else**:

print "File1 is being imported"

# File2.py

**import** File1

**print** "File2 \_\_name\_\_ = %s" **%**\_\_name\_\_

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    print "File2 is being run directly"

**else**:

print "File2 is being imported"

**OUTPUT:**

Now the interpreter is given the command to run File1.py.

**python File1.py**

Output :

File1 \_\_name\_\_ = \_\_main\_\_

File1 is being run directly

And then File2.py is run.

**python File2.py**

Output :

File1 \_\_name\_\_ = File1

File1 is being imported

File2 \_\_name\_\_ = \_\_main\_\_

File2 is being run directly