UNIT I INTRODUCTION TO PYTHON

Python installation, Python syntax, Scripts, Native Data Types, Booleans, Numbers, Lists, Tuple, Sets, Dictionaries, Comprehensions, List Comprehensions, Dictionary Comprehensions, Set Comprehensions

Python - Syntax

The Python syntax defines a set of rules that are used to create a Python Program. The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages. However, there are some definite differences between the languages.

First Python Program

Let us execute a [Python program to print "Hello, World!"](https://www.tutorialspoint.com/python/python_hello_world.htm) in two different modes of Python Programming. (a) Interactive Mode Programming (b) Script Mode Programming.

Python - Interactive Mode Programming

We can invoke a [Python interpreter](https://www.tutorialspoint.com/python/online-python-compiler.php) from command line by typing **python** at the command prompt as following −

$ python3

Python 3.10.6 (main, Mar 10 2023, 10:55:28) [GCC 11.3.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>

Here **>>>** denotes a Python Command Prompt where you can type your commands. Let's type the following text at the Python prompt and press the Enter −

>>> print ("Hello, World!")

If you are running older version of Python, like Python 2.4.x, then you would need to use print statement without parenthesis as in **print "Hello, World!"**. However in Python version 3.x, this produces the following result −

Hello, World!

Python - Script Mode Programming

We can invoke the [Python interpreter](https://www.tutorialspoint.com/python/python_interpreter.htm) with a script parameter which begins the execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script which is simple text file. Python files have extension **.py**. Type the following source code in a **test.py** file −

print ("Hello, World!")

We assume that you have Python interpreter [path set in PATH variable](https://www.tutorialspoint.com/python/python_environment.htm). Now, let's try to run this program as follows −

$ python3 test.py

This produces the following result −

Hello, World!

Let us try another way to execute a Python script. Here is the modified test.py file −

#!/usr/bin/python3

print ("Hello, World!")

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows −

$ chmod +x test.py # This is to make file executable

$./test.py

This produces the following result −

Hello, World!

Python Identifiers

A Python identifier is a name used to identify a [variable](https://www.tutorialspoint.com/python/python_variables.htm), [function](https://www.tutorialspoint.com/python/python_functions.htm), [class](https://www.tutorialspoint.com/python/python_object_classes.htm), [module](https://www.tutorialspoint.com/python/python_modules.htm) or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers.

*Python is a case sensitive programming language. Thus,****Manpower****and****manpower****are two different identifiers in Python.*

Here are naming conventions for Python identifiers −

* Python Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
* Starting an identifier with a single leading underscore indicates that the identifier is **private** identifier.
* Starting an identifier with two leading underscores indicates a strongly **private** identifier.
* If the identifier also ends with two trailing underscores, the identifier is a **language-defined** special name.

Python Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

|  |  |  |
| --- | --- | --- |
| And | as | assert |
| Break | class | continue |
| Def | del | elif |
| Else | except | False |
| finally | for | from |
| global | if | import |
| In | is | lambda |
| None | nonlocal | not |
| Or | pass | raise |
| Return | True | try |
| While | with | yield |

Python Lines and Indentation

Python programming provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by **line indentation**, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

if True:

print ("True")

else:

print ("False")

However, the following block generates an error −

if True:

print ("Answer")

print ("True")

else:

print ("Answer")

print ("False")

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks −

*Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.*

import sys

try:

# open file stream

file = open(file\_name, "w")

except IOError:

print "There was an error writing to", file\_name

sys.exit()

print "Enter '", file\_finish,

print "' When finished"

while file\_text != file\_finish:

file\_text = raw\_input("Enter text: ")

if file\_text == file\_finish:

# close the file

file.close

break

file.write(file\_text)

file.write("\n")

file.close()

file\_name = raw\_input("Enter filename: ")

if len(file\_name) == 0:

print "Next time please enter something"

sys.exit()

try:

file = open(file\_name, "r")

except IOError:

print "There was an error reading file"

sys.exit()

file\_text = file.read()

file.close()

print file\_text

Python Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

total = item\_one + \

item\_two + \

item\_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example following statement works well in Python −

days = ['Monday', 'Tuesday', 'Wednesday',

'Thursday', 'Friday']

Quotations in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

word = 'word'

print (word)

sentence = "This is a sentence."

print (sentence)

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""

print (paragraph)

Comments in Python

A comment is a programmer-readable explanation or annotation in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter

Just like most modern languages, Python supports single-line (or end-of-line) and multi-line (block) comments. [Python comments](https://www.tutorialspoint.com/python/python_comments.htm) are very much similar to the comments available in PHP, BASH and Perl Programming languages.

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

# First comment

print ("Hello, World!") # Second comment

This produces the following result −

Hello, World!

You can type a comment on the same line after a statement or expression −

name = "Madisetti" # This is again comment

You can comment multiple lines as follows −

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

'''

This is a multiline

comment.

'''

Using Blank Lines in Python Programs

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt, the statement saying “Press the enter key to exit”, and waits for the user to take action −

#!/usr/bin/python

raw\_input("\n\nPress the enter key to exit.")

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon −

import sys; x = 'foo'; sys.stdout.write(x + '\n')

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite. For example −

if expression :

suite

elif expression :

suite

else :

suite

Command Line Arguments in Python

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h −

$ python3 -h

usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...

Options and arguments (and corresponding environment variables):

-c cmd : program passed in as string (terminates option list)

-d : debug output from parser (also PYTHONDEBUG=x)

-E : ignore environment variables (such as PYTHONPATH)

-h : print this help message and exit

[ etc. ]

You can also program your script in such a way that it should accept various options. [Command Line Arguments](https://www.tutorialspoint.com/python/python_command_line_arguments.htm) is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

Scripts

A Python script or program is a file containing executable Python code. Being able to run Python scripts and code is probably the most important skill that you need as a Python developer. By running your code, you'll know if it works as planned. You'll also be able to test and debug the code to fix errors and issues

Native Data Types



Lists

**Python Lists**are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [ ] and separated by commas.

*The list is a sequence data type which is used to store the collection of data.*[*Tuples*](https://www.geeksforgeeks.org/python-tuples/)*and*[*String*](https://www.geeksforgeeks.org/python-strings/)*are other types of sequence data types.*

## ****Example of list in Python****

Here we are creating Python **List** using [].

* Python3

|  |
| --- |
| Var **=** ["Geeks", "for", "Geeks"]  print(Var) |

**Output:**

["Geeks", "for", "Geeks"]

Lists are the simplest containers that are an integral part of the Python language. Lists need not be homogeneous always which makes it the most powerful tool in [Python](https://www.geeksforgeeks.org/python-programming-language/). A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

## Creating a List in Python

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike [Sets](https://www.geeksforgeeks.org/python-sets/), a list doesn’t need a built-in function for its creation of a list.

***Note:****Unlike Sets, the list may contain mutable elements.*

### Example 1: Creating a list in Python

* Python3

|  |
| --- |
| # Python program to demonstrate  # Creation of List    # Creating a List  List **=** []  **print**("Blank List: ")  **print**(List)    # Creating a List of numbers  List **=** [10, 20, 14]  print("\nList of numbers: ")  print(List)    # Creating a List of strings and accessing  # using index  List **=** ["Geeks", "For", "Geeks"]  print("\nList Items: ")  print(List[0])  print(List[2]) |

**Output**

Blank List:

[]

List of numbers:

[10, 20, 14]

List Items:

Geeks

Geeks

#### Complexities for Creating Lists

**Time Complexity:**O(1)

**Space Complexity:**O(n)

## Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

**Example 1: Accessing elements from list**

* Python3

|  |
| --- |
| # Python program to demonstrate  # accessing of element from list    # Creating a List with  # the use of multiple values  List **=** ["Geeks", "For", "Geeks"]    # accessing a element from the  # list using index number  print("Accessing a element from the list")  **print**(List[0])  print(List[2]) |

**Output**

Accessing a element from the list

Geeks

Geeks

## Getting the size of Python list

Python[len()](https://www.geeksforgeeks.org/python-ways-to-find-length-of-list/) is used to get the length of the list.

* Python3

|  |
| --- |
| # Creating a List  List1 **=** []  print(len(List1))    # Creating a List of numbers  List2 **=** [10, 20, 14]  print(len(List2)) |

**Output**

0

3

## Taking Input of a Python List

We can take the input of a list of elements as string, integer, float, etc. But the default one is a string.

**Example 1:**

* Python3

|  |
| --- |
| # Python program to take space  # separated input as a string  # split and store it to a list  # and print the string list    # input the list as string  string **=** input("Enter elements (Space-Separated): ")    # split the strings and store it to a list  lst **=** string.split()  print('The list is:', lst)   # printing the list |

**Output:**

Enter elements: GEEKS FOR GEEKS

The list is: ['GEEKS', 'FOR', 'GEEKS']

**Example 2:**

* Python

|  |
| --- |
| # input size of the list  n **=** int(input("Enter the size of list : "))  # store integers in a list using map,  # split and strip functions  lst **=** list(map(int, input("Enter the integer\  elements:").strip().split()))[:n]    # printing the list  print('The list is:', lst) |

**Output:**

Enter the size of list : 4

Enter the integer elements: 6 3 9 10

The list is: [6, 3, 9, 10]

To know more see [this](https://www.geeksforgeeks.org/python-get-a-list-as-input-from-user/).

## Adding Elements to a Python List

### Method 1: Using append() method

Elements can be added to the List by using the built-in [**append()**](https://www.geeksforgeeks.org/list-methods-python/) function. Only one element at a time can be added to the list by using the append() method, for the addition of multiple elements with the append() method, loops are used. Tuples can also be added to the list with the use of the append method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the append() method.

* Python3

|  |
| --- |
| # Python program to demonstrate  # Addition of elements in a List    # Creating a List  List **=** []  **print**("Initial blank List: ")  **print**(List)    # Addition of Elements  # in the List  List.append(1)  List.append(2)  List.append(4)  **print**("\nList after Addition of Three elements: ")  **print**(List)    # Adding elements to the List  # using Iterator  **for** i **in** range(1, 4):      List.append(i)  **print**("\nList after Addition of elements from 1-3: ")  print(List)    # Adding Tuples to the List  List.append((5, 6))  **print**("\nList after Addition of a Tuple: ")  print(List)    # Addition of List to a List  List2 **=** ['For', 'Geeks']  List.append(List2)  **print**("\nList after Addition of a List: ")  print(List) |

**Output**

Initial blank List:

[]

List after Addition of Three elements:

[1, 2, 4]

List after Addition of elements from 1-3:

[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:

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

List after Addition of a List:

[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]

#### Complexities for Adding elements in a Lists(append() method):

**Time Complexity:** O(1)

S**pace Complexity:** O(1)

### Method 2: Using insert() method

append() method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, [insert()](https://www.geeksforgeeks.org/python-list-insert/)method is used. Unlike append() which takes only one argument, the insert() method requires two arguments(position, value).

* Python3

|  |
| --- |
| # Python program to demonstrate  # Addition of elements in a List    # Creating a List  List **=** [1,2,3,4]  **print**("Initial List: ")  print(List)    # Addition of Element at  # specific Position  # (using Insert Method)  List.insert(3, 12)  List.insert(0, 'Geeks')  print("\nList after performing Insert Operation: ")  **print**(List) |

**Output**

Initial List:

[1, 2, 3, 4]

List after performing Insert Operation:

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

#### Complexities for Adding elements in a Lists(insert() method):

**Time Complexity:** O(n)

**Space Complexity:** O(1)

### Method 3: Using extend() method

Other than append() and insert() methods, there’s one more method for the Addition of elements, [**extend()**](https://www.geeksforgeeks.org/append-extend-python/), this method is used to add multiple elements at the same time at the end of the list.

***Note:***[*append() and extend()*](https://www.geeksforgeeks.org/append-extend-python/)*methods can only add elements at the end.*

* Python3

|  |
| --- |
| # Python program to demonstrate  # Addition of elements in a List    # Creating a List  List **=** [1, 2, 3, 4]  **print**("Initial List: ")  **print**(List)    # Addition of multiple elements  # to the List at the end  # (using Extend Method)  List.extend([8, 'Geeks', 'Always'])  print("\nList after performing Extend Operation: ")  print(List) |

**Output**

Initial List:

[1, 2, 3, 4]

List after performing Extend Operation:

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

#### Complexities for Adding elements in a Lists(extend() method):

**Time Complexity:** O(n)

**Space Complexity:** O(1)

## Removing Elements from the List

### Method 1: Using remove() method

Elements can be removed from the List by using the built-in [**remove()**](https://www.geeksforgeeks.org/python-list-remove/) functionbut an Error arises if the element doesn’t exist in the list. Remove() method only removes one element at a time, to remove a range of elements, theiterator is used. The remove() method removes the specified item.

***Note:****Remove method in List will only remove the first occurrence of the searched element.*

**Example 1:**

* Python3

|  |
| --- |
| # Python program to demonstrate  # Removal of elements in a List    # Creating a List  List **=** [1, 2, 3, 4, 5, 6,          7, 8, 9, 10, 11, 12]  **print**("Initial List: ")  **print**(List)    # Removing elements from List  # using Remove() method  List.remove(5)  List.remove(6)  print("\nList after Removal of two elements: ")  print(List) |

**Output**

Initial List:

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

List after Removal of two elements:

[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

**Example 2:**

* Python3

|  |
| --- |
| # Creating a List  List **=** [1, 2, 3, 4, 5, 6,          7, 8, 9, 10, 11, 12]  # Removing elements from List  # using iterator method  **for** i **in** range(1, 5):      List.remove(i)  print("\nList after Removing a range of elements: ")  print(List) |

**Output**

List after Removing a range of elements:

[5, 6, 7, 8, 9, 10, 11, 12]

## Slicing of a List

We can get substrings and sublists using a slice. 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 the [Slice operation](https://www.geeksforgeeks.org/python-list-comprehension-and-slicing/).

Slice operation is performed on Lists with the use of a colon(:)

**To print elements from beginning to a range use:**

*[: Index]*

To print elements from end-use:

*[:-Index]*

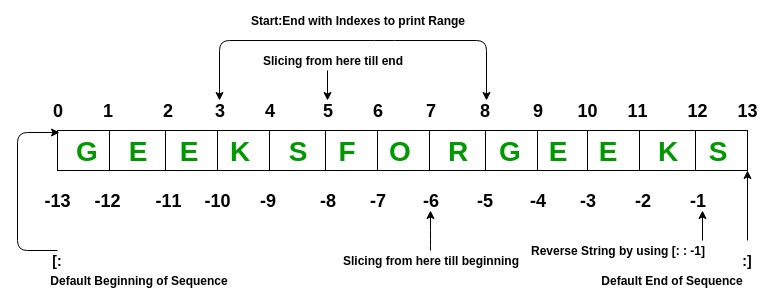
To print elements from a specific Index till the end use

*[Index:]*

To print the whole list in reverse order, use

*[::-1]*

**Note –**To print elements of List from rear-end, use Negative Indexes.



**UNDERSTANDING SLICING OF LISTS:**

* pr[0] accesses the first item, 2.
* pr[-4] accesses the fourth item from the end, 5.
* pr[2:] accesses [5, 7, 11, 13], a list of items from third to last.
* pr[:4] accesses [2, 3, 5, 7], a list of items from first to fourth.
* pr[2:4] accesses [5, 7], a list of items from third to fifth.
* pr[1::2] accesses [3, 7, 13], alternate items, starting from the second item.
* Python3

|  |
| --- |
| # Python program to demonstrate  # Removal of elements in a List    # Creating a List  List **=** ['G', 'E', 'E', 'K', 'S', 'F',          'O', 'R', 'G', 'E', 'E', 'K', 'S']  **print**("Initial List: ")  **print**(List)    # Print elements of a range  # using Slice operation  Sliced\_List **=** List[3:8]  print("\nSlicing elements in a range 3-8: ")  **print**(Sliced\_List)    # Print elements from a  # pre-defined point to end  Sliced\_List **=** List[5:]  print("\nElements sliced from 5th "        "element till the end: ")  print(Sliced\_List)    # Printing elements from  # beginning till end  Sliced\_List **=** List[:]  print("\nPrinting all elements using slice operation: ")  print(Sliced\_List) |

**Output**

Initial List:

['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:

['K', 'S', 'F', 'O', 'R']

Elements sliced from 5th element till the end:

['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:

['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

## List Comprehension

[**Python List comprehensions**](https://www.geeksforgeeks.org/python-list-comprehension/) are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc. A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

**Syntax:**

*newList = [ expression(element) for element in oldList if condition ]*

**Example:**

* Python3

|  |
| --- |
| # 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) |

**Output**

[1, 9, 25, 49, 81]

For better understanding, the above code is similar to as follows:

* Python3

|  |
| --- |
| # 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) |

**Output**

[1, 9, 25, 49, 81]

Refer to the below articles to get detailed information about List Comprehension.

## List Methods

| **Function** | **Description** |
| --- | --- |
| [Append()](https://www.geeksforgeeks.org/append-extend-python/) | Add an element to the end of the list |
| [Extend()](https://www.geeksforgeeks.org/append-extend-python/) | Add all elements of a list to another list |
| [Insert()](https://www.geeksforgeeks.org/list-methods-in-python-set-2-del-remove-sort-insert-pop-extend/) | Insert an item at the defined index |
| [Remove()](https://www.geeksforgeeks.org/list-methods-in-python-set-2-del-remove-sort-insert-pop-extend/) | Removes an item from the list |
| [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/) | Returns the index of the first matched item |
| [Count()](https://www.geeksforgeeks.org/python-list-function-count/) | Returns the count of the number of items passed as an argument |
| [Sort()](https://www.geeksforgeeks.org/sort-in-python/) | Sort items in a list in ascending order |
| [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 |
| [copy()](https://www.geeksforgeeks.org/python-list-copy-method/) | Returns a copy of the list |
| [pop()](https://www.geeksforgeeks.org/python-list-pop-method/) | Removes and returns the item at the specified index. If no index is provided, it removes and returns the last item. |

To know more refer to this article – [Python List methods](https://www.geeksforgeeks.org/list-methods-python/)

The operations mentioned above modify the list Itself.

### Built-in functions with List

| **Function** | **Description** |
| --- | --- |
| [reduce()](https://www.geeksforgeeks.org/reduce-in-python/) | 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/) | Sums up the numbers in the list |
| [ord()](https://www.geeksforgeeks.org/ord-function-python/) | Returns an integer representing the Unicode code point of the given Unicode character |
| [cmp()](https://www.geeksforgeeks.org/python-2-number-cmplist-method/) | This function returns 1 if the first list is “greater” than the second list |
| [max()](https://www.geeksforgeeks.org/python-max-function/) | return maximum element of a given list |
| [min()](https://www.geeksforgeeks.org/python-min-function/) | return minimum element of a given list |
| [all()](https://www.geeksforgeeks.org/any-all-in-python/) | Returns true if all element is true or if the list is empty |
| [any()](https://www.geeksforgeeks.org/any-all-in-python/) | return true if any element of the list is true. if the list is empty, return false |
| [len()](https://www.geeksforgeeks.org/python-len-function/) | Returns length of the list or size of the list |
| [enumerate()](https://www.geeksforgeeks.org/enumerate-in-python/) | Returns enumerate object of the list |
| [accumulate()](https://www.geeksforgeeks.org/python-itertools-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/) | tests if each element of a list is true or not |
| [map()](https://www.geeksforgeeks.org/python-map-function/) | 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/) | This function can have any number of arguments but only one expression, which is evaluated and returned. |

**Tuple**

# Python Tuples

**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.

Values of a tuple are syntactically separated by ‘commas’. Although it is not necessary, it is more common to define a tuple by closing the sequence of values in parentheses. This helps in understanding the Python tuples more easily.

## Creating a Tuple

In Python, tuples are created by placing a sequence of values separated by ‘comma’ with or without the use of parentheses for grouping the data sequence.

**Note:**Creation of Python tuple without the use of parentheses is known as Tuple Packing.

#### Python program to demonstrate the addition of elements in a Tuple.

* Python3

|  |
| --- |
| # Creating an empty Tuple  Tuple1 **=** ()  print("Initial empty Tuple: ")  print(Tuple1)    # Creating a Tuple  # with the use of string  Tuple1 **=** ('Geeks', 'For')  print("\nTuple with the use of String: ")  print(Tuple1)    # Creating a Tuple with  # the use of list  list1 **=** [1, 2, 4, 5, 6]  print("\nTuple using List: ")  **print**(tuple(list1))    # Creating a Tuple  # with the use of built-in function  Tuple1 **=** tuple('Geeks')  print("\nTuple with the use of function: ")  print(Tuple1) |

**Output:**

Initial empty Tuple:

()

Tuple with the use of String:

('Geeks', 'For')

Tuple using List:

(1, 2, 4, 5, 6)

Tuple with the use of function:

('G', 'e', 'e', 'k', 's')

#### Creating a Tuple with Mixed Datatypes.

**Tuples**can contain any number of elements and of any datatype (like strings, integers, list, etc.). Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing ‘comma’ to make it a tuple.

* Python3

|  |
| --- |
| # Creating a Tuple  # with Mixed Datatype  Tuple1 **=** (5, 'Welcome', 7, 'Geeks')  **print**("\nTuple with Mixed Datatypes: ")  **print**(Tuple1)    # Creating a Tuple  # with nested tuples  Tuple1 **=** (0, 1, 2, 3)  Tuple2 **=** ('python', 'geek')  Tuple3 **=** (Tuple1, Tuple2)  **print**("\nTuple with nested tuples: ")  print(Tuple3)    # Creating a Tuple  # with repetition  Tuple1 **=** ('Geeks',) **\*** 3  **print**("\nTuple with repetition: ")  **print**(Tuple1)    # Creating a Tuple  # with the use of loop  Tuple1 **=** ('Geeks')  n **=** 5  print("\nTuple with a loop")  **for** i **in** range(int(n)):      Tuple1 **=** (Tuple1,)      print(Tuple1) |

**Output:**

Tuple with Mixed Datatypes:

(5, 'Welcome', 7, 'Geeks')

Tuple with nested tuples:

((0, 1, 2, 3), ('python', 'geek'))

Tuple with repetition:

('Geeks', 'Geeks', 'Geeks')

Tuple with a loop

('Geeks',)

(('Geeks',),)

((('Geeks',),),)

(((('Geeks',),),),)

((((('Geeks',),),),),)

### Complexities for creating tuples:

**Time complexity:** O(1)

**Auxiliary Space :** O(n)

## Accessing of Tuples

**Tuples** are immutable, and usually, they contain a sequence of heterogeneous elements that are accessed via [unpacking](https://www.geeksforgeeks.org/unpacking-a-tuple-in-python/)or indexing (or even by attribute in the case of named tuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

**Note:**In unpacking of tuple number of variables on the left-hand side should be equal to a number of values in given tuple a.

* Python3

|  |
| --- |
| # Accessing Tuple  # with Indexing  Tuple1 **=** tuple("Geeks")  **print**("\nFirst element of Tuple: ")  print(Tuple1[0])      # Tuple unpacking  Tuple1 **=** ("Geeks", "For", "Geeks")    # This line unpack  # values of Tuple1  a, b, c **=** Tuple1  print("\nValues after unpacking: ")  **print**(a)  print(b)  print(c) |

**Output:**

First element of Tuple:

G

Values after unpacking:

Geeks

For

Geeks

### Complexities for accessing elements in tuples:

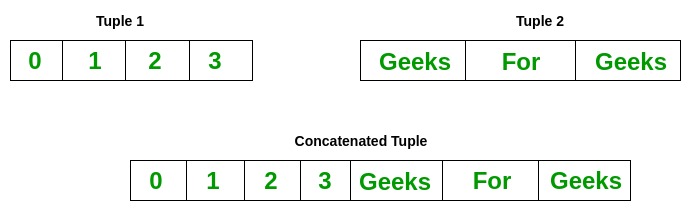
**Time complexity:**O(1)

**Space complexity:**O(1)

## 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.



* Python3

|  |
| --- |
| # 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')

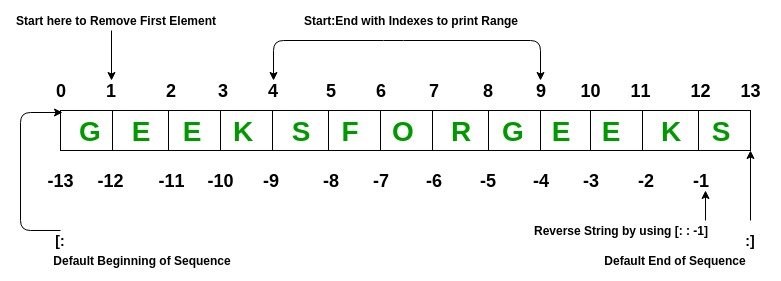
**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

## Slicing of Tuple

Slicing of a Tuple is done to fetch a specific range or slice of sub-elements from a Tuple. Slicing can also be done to lists and arrays. Indexing in a list results to fetching a single element whereas Slicing allows to fetch a set of elements.

**Note-** Negative Increment values can also be used to reverse the sequence of Tuples.



* Python3

|  |
| --- |
| # Slicing of a Tuple    # Slicing of a Tuple  # with Numbers  Tuple1 **=** tuple('GEEKSFORGEEKS')    # Removing First element  print("Removal of First Element: ")  **print**(Tuple1[1:])    # Reversing the Tuple  **print**("\nTuple after sequence of Element is reversed: ")  **print**(Tuple1[::**-**1])    # Printing elements of a Range  print("\nPrinting elements between Range 4-9: ")  print(Tuple1[4:9]) |

**Output:**

Removal of First Element:

('E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S')

Tuple after sequence of Element is reversed:

('S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G')

Printing elements between Range 4-9:

('S', 'F', 'O', 'R', 'G')

### Complexities for traversal/searching elements in tuples:

**Time complexity:** O(1)

**Space complexity:** O(1)

## Deleting a Tuple

Tuples are immutable and hence they do not allow deletion of a part of it. The entire tuple gets deleted by the use of del() method.

**Note-** Printing of Tuple after deletion results in an Error.

* Python

|  |
| --- |
| # Deleting a Tuple    Tuple1 **=** (0, 1, 2, 3, 4)  **del** Tuple1    print(Tuple1) |

*Traceback (most recent call last):   
File “/home/efa50fd0709dec08434191f32275928a.py”, line 7, in   
print(Tuple1)   
NameError: name ‘Tuple1’ is not defined*

#### Built-In Methods

| **Built-in-Method** | **Description** |
| --- | --- |
| [**index( )**](https://www.geeksforgeeks.org/python-list-index/) | Find in the tuple and returns the index of the given value where it’s available |
| [**count( )**](https://www.geeksforgeeks.org/python-list-count-method/) | Returns the frequency of occurrence of a specified value |

#### Built-In Functions

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

### 

### Tuples VS Lists:

|  |  |
| --- | --- |
| **Similarities** | **Differences** |
| Functions that can be used for both lists and tuples:  len(), max(), min(), sum(), any(), all(), sorted() | Methods that cannot be used for tuples:  append(), insert(), remove(), pop(), clear(), sort(), reverse() |
| Methods that can be used for both lists and tuples:  count(), Index() | we generally use ‘tuples’ for heterogeneous (different) data types and ‘lists’ for homogeneous (similar) data types. |
| Tuples can be stored in lists. | Iterating through a ‘tuple’ is faster than in a ‘list’. |
| Lists can be stored in tuples. | ‘Lists’ are mutable whereas ‘tuples’ are immutable. |
| Both ‘tuples’ and ‘lists’ can be nested. | Tuples that contain immutable elements can be used as a key for a dictionary. |

Sets

# Python Sets

In Python, a **Set**is an unordered collection of data types 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.

### Creating a set with another method

* Python3

|  |
| --- |
| # Another Method to create sets in Python3    # Set containing numbers  my\_set **=** {1, 2, 3}    print(my\_set)    # This code is contributed by sarajadhav12052009 |

**Output**

{1, 2, 3}

### 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.

* Python3

|  |
| --- |
| # 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**("\n")  print("Geeks" **in** set1) |

**Output**

Initial set

{'Geeks.', 'For', 'Geeks'}

Elements of set:

Geeks. For Geeks

True

### Using 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.*

* Python3

|  |
| --- |
| # 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 element from the  # Set using the pop() method  set1.pop()  print("\nSet after popping an element: ")  print(set1) |

**Output**

Initial Set:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Set after popping an element:

{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

### Example: Implementing all functions:

* Python3

|  |
| --- |
| **def** create\_set():      my\_set **=** {1, 2, 3, 4, 5}  **print**(my\_set)    **def** add\_element():      my\_set **=** {1, 2, 3, 4, 5}      my\_set.add(6)  **print**(my\_set)    **def** remove\_element():      my\_set **=** {1, 2, 3, 4, 5}      my\_set.remove(3)  **print**(my\_set)    **def** clear\_set():      my\_set **=** {1, 2, 3, 4, 5}      my\_set.clear()  **print**(my\_set)    **def** set\_union():      set1 **=** {1, 2, 3}      set2 **=** {4, 5, 6}      my\_set **=** set1.union(set2)  **print**(my\_set)    **def** set\_intersection():      set1 **=** {1, 2, 3, 4, 5}      set2 **=** {4, 5, 6, 7, 8}      my\_set **=** set1.intersection(set2)      print(my\_set)    **def** set\_difference():      set1 **=** {1, 2, 3, 4, 5}      set2 **=** {4, 5, 6, 7, 8}      my\_set **=** set1.difference(set2)  **print**(my\_set)    **def** set\_symmetric\_difference():      set1 **=** {1, 2, 3, 4, 5}      set2 **=** {4, 5, 6, 7, 8}      my\_set **=** set1.symmetric\_difference(set2)      print(my\_set)    **def** set\_subset():      set1 **=** {1, 2, 3, 4, 5}      set2 **=** {2, 3, 4}      subset **=** set2.issubset(set1)  **print**(subset)    **def** set\_superset():      set1 **=** {1, 2, 3, 4, 5}      set2 **=** {2, 3, 4}      superset **=** set1.issuperset(set2)      print(superset)    **if** \_\_name\_\_ **==** '\_\_main\_\_':      create\_set()      add\_element()      remove\_element()      clear\_set()      set\_union()      set\_intersection()      set\_difference()      set\_symmetric\_difference()      set\_subset()      set\_superset() |

**Output**

{1, 2, 3, 4, 5}

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

{1, 2, 4, 5}

set()

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

{4, 5}

{1, 2, 3}

{1, 2, 3, 6, 7, 8}

True

True

### Advantages:

* **Unique Elements**: Sets can only contain unique elements, so they can be useful for removing duplicates from a collection of data.
* **Fast Membership Testing**: Sets are optimized for fast membership testing, so they can be useful for determining whether a value is in a collection or not.
* **Mathematical Set Operations:** Sets support mathematical set operations like union, intersection, and difference, which can be useful for working with sets of data.
* **Mutable**: Sets are mutable, which means that you can add or remove elements from a set after it has been created.

### Disadvantages:

* **Unordered**: Sets are unordered, which means that you cannot rely on the order of the data in the set. This can make it difficult to access or process data in a specific order.
* **Limited Functionality:** Sets have limited functionality compared to lists, as they do not support methods like append() or pop(). This can make it more difficult to modify or manipulate data stored in a set.
* **Memory Usage:** Sets can consume more memory than lists, especially for small datasets. This is because each element in a set requires additional memory to store a hash value.
* **Less Commonly Used:** Sets are less commonly used than lists and dictionaries in Python, which means that there may be fewer resources or libraries available for working with them. This can make it more difficult to find solutions to problems or to get help with debugging.

Overall, sets can be a useful data structure in Python, especially for removing duplicates or for fast membership testing. However, their lack of ordering and limited functionality can also make them less versatile than lists or dictionaries, so it is important to carefully consider the advantages and disadvantages of using sets when deciding which data structure to use in your Python program.

### Set Methods

| **Function** | **Description** |
| --- | --- |
| [add()](https://www.geeksforgeeks.org/set-add-python/) | Adds an element to a set |
| [remove()](https://www.geeksforgeeks.org/python-remove-discard-sets/) | Removes an element from a set. If the element is not present in the set, raise a KeyError |
| [clear()](https://www.geeksforgeeks.org/set-clear-python/) | Removes all elements form a set |
| [copy()](https://www.geeksforgeeks.org/set-copy-python/) | Returns a shallow copy of a set |
| [pop()](https://www.geeksforgeeks.org/python-set-pop/) | Removes and returns an arbitrary set element. Raise KeyError if the set is empty |
| [update()](https://www.geeksforgeeks.org/python-set-update/) | Updates a set with the union of itself and others |
| [union()](https://www.geeksforgeeks.org/union-function-python/) | Returns the union of sets in a new set |
| [difference()](https://www.geeksforgeeks.org/python-set-difference/) | Returns the difference of two or more sets as a new set |
| [difference\_update()](https://www.geeksforgeeks.org/python-set-difference_update/) | Removes all elements of another set from this set |
| [discard()](https://www.geeksforgeeks.org/python-remove-discard-sets/) | Removes an element from set if it is a member. (Do nothing if the element is not in set) |
| [intersection()](https://www.geeksforgeeks.org/intersection-function-python/) | Returns the intersection of two sets as a new set |
| intersection\_update() | Updates the set with the intersection of itself and another |
| [isdisjoint()](https://www.geeksforgeeks.org/isdisjoint-function-python/) | Returns True if two sets have a null intersection |
| [issubset()](https://www.geeksforgeeks.org/issubset-in-python/) | Returns True if another set contains this set |
| [issuperset()](https://www.geeksforgeeks.org/issuperset-in-python/) | Returns True if this set contains another set |
| [symmetric\_difference()](https://www.geeksforgeeks.org/python-set-symmetric_difference-2/) | Returns the symmetric difference of two sets as a new set |
| [symmetric\_difference\_update()](https://www.geeksforgeeks.org/python-set-symmetric_difference_update/) | Updates a set with the symmetric difference of itself and another |

Dictionaries

**What is a Python Dictionary?**

Dictionaries in Python is a data structure, used to store values in key:value format. This makes it different from lists, tuples, and arrays as in a dictionary each key has an associated value.

***Note:****As of Python version 3.7, dictionaries are ordered and can not contain duplicate keys.*

**How to Create a Dictionary**

In [Python](https://www.geeksforgeeks.org/python-programming-language/), a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by a ‘comma’.

The dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**.

Values in a dictionary can be of any data type and can be duplicated, whereas keys can’t be repeated and must be *immutable*.

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

The code demonstrates creating dictionaries with different types of keys. The first dictionary uses integer keys, and the second dictionary uses a mix of string and integer keys with corresponding values. This showcases the flexibility of Python dictionaries in handling various data types as keys.

* Python3

|  |
| --- |
| Dict **=** {1: 'Geeks', 2: 'For', 3: 'Geeks'}  print("\nDictionary with the use of Integer Keys: ")  **print**(Dict)    Dict **=** {'Name': 'Geeks', 1: [1, 2, 3, 4]}  print("\nDictionary with the use of Mixed Keys: ")  print(Dict) |

**Output**

Dictionary with the use of Integer Keys:   
{1: 'Geeks', 2: 'For', 3: 'Geeks'}  
Dictionary with the use of Mixed Keys:   
{'Name': 'Geeks', 1: [1, 2, 3, 4]}

**Dictionary Methods**

Here is a list of in-built dictionary functions with their description. You can use these functions to operate on a dictionary.

| **Method** | **Description** |
| --- | --- |
| dict.clear() | Remove all the elements from the dictionary |
| dict.copy() | Returns a copy of the dictionary |
| dict.get(key, default = “None”) | Returns the value of specified key |
| dict.items() | Returns a list containing a tuple for each key value pair |
| dict.keys() | Returns a list containing dictionary’s keys |
| dict.update(dict2) | Updates dictionary with specified key-value pairs |
| dict.values() | Returns a list of all the values of dictionary |
| pop() | Remove the element with specified key |
| popItem() | Removes the last inserted key-value pair |
| dict.setdefault(key,default= “None”) | set the key to the default value if the key is not specified in the dictionary |
| dict.has\_key(key) | returns true if the dictionary contains the specified key. |
| dict.get(key, default = “None”) | used to get the value specified for the passed key. |

***For Detailed Explanations:***[***Python Dictionary Methods***](https://www.geeksforgeeks.org/python-dictionary-methods/)

**Multiple Dictionary Operations in Python**

The code begins with a dictionary **‘dict1’** and creates a copy **‘dict2’**. It then demonstrates several dictionary operations: clearing **‘dict1’**, accessing values, retrieving key-value pairs and keys, removing specific key-value pairs, updating a value, and retrieving values. These operations showcase how to work with dictionaries in Python.

* Python3

|  |
| --- |
| dict1 **=** {1: "Python", 2: "Java", 3: "Ruby", 4: "Scala"}  dict2 **=** dict1.copy()  **print**(dict2)  dict1.clear()  **print**(dict1)  **print**(dict2.get(1))  print(dict2.items())  print(dict2.keys())  dict2.pop(4)  print(dict2)  dict2.popitem()  print(dict2)  dict2.update({3: "Scala"})  print(dict2)  print(dict2.values()) |

**Output:**

{1: 'Python', 2: 'Java', 3: 'Ruby', 4: 'Scala'}  
{}  
Python  
dict\_items([(1, 'Python'), (2, 'Java'), (3, 'Ruby'), (4, 'Scala')])  
dict\_keys([1, 2, 3, 4])  
{1: 'Python', 2: 'Java', 3: 'Ruby'}  
{1: 'Python', 2: 'Java'}  
{1: 'Python', 2: 'Java', 3: 'Scala'}  
dict\_values(['Python', 'Java', 'Scala']

Comprehensions

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, sets, dictionaries, etc.) using previously defined sequences. [Python](https://www.geeksforgeeks.org/python-programming-language/) supports the following 4 types of comprehension:

* List Comprehensions
* Dictionary Comprehensions
* Set Comprehensions
* Generator Comprehensions

**List Comprehensions**

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of list comprehension:

***Syntax:****output\_list = [output\_exp for var in input\_list if (var satisfies this condition)]*

Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple

**Example 1: Generating an Even list WITHOUT using List comprehensions**

0 seconds of 0 secondsVolume 0%

Suppose we want to create an output list that contains only the even numbers which are present in the input list. Let’s see how to do this using *loops, list comprehension,* and *list comprehension,* and decide which method suits you better.

* Python3

|  |
| --- |
| input\_list **=** [1, 2, 3, 4, 4, 5, 6, 7, 7]  output\_list **=** []    **for** var **in** input\_list:  **if** var **%** 2 **==** 0:          output\_list.append(var)    print("Output List using for loop:", output\_list) |

**Output:**

Output List using for loop: [2, 4, 4, 6]

**Example 2:** **Generating Even list using List comprehensions**

Here we use the**list comprehensions**in Python. It creates a new list named list\_using\_comp by iterating through each element var in the input\_list. Elements are included in the new list only if they satisfy the condition, which checks if the element is even. As a result, the output list will contain all even numbers.

* Python3

|  |
| --- |
| input\_list **=** [1, 2, 3, 4, 4, 5, 6, 7, 7]    list\_using\_comp **=** [var **for** var **in** input\_list **if** var **%** 2 **==** 0]    print("Output List using list comprehensions:",                                 list\_using\_comp) |

## Dictionary Comprehensions

Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.

*output\_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}*

**Example 1:** **Generating odd number with their cube values without using dictionary comprehension**

Suppose we want to create an output dictionary which contains only the odd numbers that are present in the input list as keys and their cubes as values. Let’s see how to do this using for loops and dictionary comprehension.

* Python3

|  |
| --- |
| input\_list **=** [1, 2, 3, 4, 5, 6, 7]    output\_dict **=** {}    **for** var **in** input\_list:  **if** var **%** 2 !**=** 0:          output\_dict[var] **=** var**\*\***3    print("Output Dictionary using for loop:",output\_dict ) |

**Output:**

Output Dictionary using for loop: {1: 1, 3: 27, 5: 125, 7: 343}

**Example 2:** **Generating odd number with their cube values** **with** **using dictionary comprehension**

We are using dictionary comprehension in Python. It initializes an list containing numbers from 1 to 7. It then constructs a new dictionary using dictionary comprehension. For each odd number var in the list, it calculates the cube of the number and assigns the result as the value to the key var in the dictionary.

* Python3

|  |
| --- |
| input\_list **=** [1,2,3,4,5,6,7]    dict\_using\_comp **=** {var:var **\*\*** 3 **for** var **in** input\_list **if** var **%** 2 !**=** 0}    print("Output Dictionary using dictionary comprehensions:",dict\_using\_comp) |

**Output:**

Output Dictionary using dictionary comprehensions: {1: 1, 3: 27, 5: 125, 7: 343}

**Set Comprehensions**

Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }

Let’s look at the following example to understand set comprehensions.

**Example 1 :** **Checking Even number Without using set comprehension**

Suppose we want to create an output set which contains only the even numbers that are present in the input list. Note that set will discard all the duplicate values. Let’s see how we can do this using for loops and set comprehension.

* Python3

|  |
| --- |
| input\_list **=** [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]    output\_set **=** set()    **for** var **in** input\_list:  **if** var **%** 2 **==** 0:          output\_set.add(var)    print("Output Set using for loop:", output\_set) |

**Output:**

Output Set using for loop: {2, 4, 6}

**Example 2: Checking Even number using set comprehension**

We will use set comprehension to initializes a list with integer values. The code then creates a new set using set comprehension. It iterates through the elements of the input\_list, and for each element, it checks whether it’s even. If the condition is met, the element is added to the set. The printed output which will contain unique even numbers from the list.

* Python3

|  |
| --- |
| input\_list **=** [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]    set\_using\_comp **=** {var **for** var **in** input\_list **if** var **%** 2 **==** 0}    print("Output Set using set comprehensions:",                                set\_using\_comp) |

**Output:**

Output Set using set comprehensions: {2, 4, 6}

Set Comprehension

Set is used to display the unique elements from a given collection. Let's obtain the square of all elements of list using set.

**Example - 1**

1. list1=[2,3,4,5,6,7,8,9,10]
2. result\_set=set()
3. for i in list1:
4. result\_set.add(i\*\*2)
5. print("The square of the numbers present in list1 is: ",result\_set)

**Output-**

The square of the numbers present in list1 is: {64, 4, 36, 100, 9, 16, 49, 81, 25}

In the program given below, we have done the same thing using comprehension.

**Example - 2- Using set comprehension**

1. list1=[2,3,4,5,6,7,8,9,10]
2. result\_set={i\*\*2 for i in list1}
3. print("The square of the numbers obtained through set comprehension: ",result\_set)

**Output-**

The square of the numbers obtained through set comprehension: {64, 4, 36, 100, 9, 16, 49, 81, 25}

We have taken each element from list1 and provided the expression in result\_set for calculating the square of these elements.

UNIT II

Strings and modules: String operation, Formatting, Bytes, Encoding, Regular Expressions, Verbose, module declaration, Importing modules, Objects, and Indenting as Requirement, Exceptions, Unbound Variables, Lambda Functions and map

Strings and modules

# Python String

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

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

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

**Example:**

"Geeksforgeeks" or 'Geeksforgeeks' or "a"

* Python3

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

**Output:**

A Computer Science portal for geeks  
A

## Creating a String in Python

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

**Example:**

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

* Python3

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

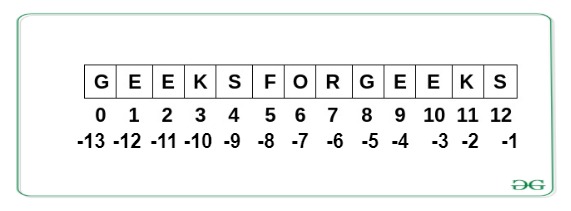
**Output:**

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

## Accessing characters in Python String

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

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



*Python String indexing*

**Example:**

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

* Python3

|  |
| --- |
| # Python Program to Access  # characters of String    String1 **=** "GeeksForGeeks"  **print**("Initial String: ")  print(String1)    # Printing First character  print("\nFirst character of String is: ")  **print**(String1[0])    # Printing Last character  **print**("\nLast character of String is: ")  print(String1[**-**1]) |

**Output:**

Initial String:   
GeeksForGeeks  
First character of String is:   
G  
Last cha racter of String is:   
s

## String Slicing

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

**Example:**

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

* Python3

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

**Output:**

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

## Reversing a Python String

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

**Example:**

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

* Python3

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

**Output:**

skeegrofskeeg

**Example:**

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

* Python3

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

**Output:**

skeegrofskeeg

## Deleting/Updating from a String

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

### Updating a character

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

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

**Example:**

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

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

* Python3

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

**Output:**

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

### Updating Entire String

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

**Example:**

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

* Python3

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

**Output:**

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

### Deleting a character

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

* Python3

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

**Output:**

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

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

**Example:**

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

* Python3

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

**Output:**

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

### Deleting Entire String

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

* Python3

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

**Error:**

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

## Escape Sequencing in Python

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

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

**Example:**

* Python3

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

**Output:**

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

**Example:**

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

* Python3

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

**Output:**

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

## Formatting of Strings

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

**Example 1:**

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

* Python3

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

**Output:**

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

String operation

n Python, a string is an ordered sequence of Unicode characters. Each character in the string has a unique index in the sequence. The index starts with 0. First character in the string has its positional index 0. The index keeps incrementing towards the end of string.

If a string variable is declared as var="HELLO PYTHON", index of each character in the string is as follows −

string_variable

Python allows you to access any individual character from the string by its index. In this case, 0 is the lower bound and 11 is the upper bound of the string. So, var[0] returns H, var[6] returns P. If the index in square brackets exceeds the upper bound, Python raises IndexError.

>>> var="HELLO PYTHON"

>>> var[0]

'H'

>>> var[7]

'Y'

>>> var[11]

'N'

>>> var[12]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: string index out of range

One of the unique features of Python sequence types (and therefore a string object) it has a negative indexing scheme also. In the example above, a positive indexing scheme is used where the index increments from left to right. In case of negative indexing, the character at the end has -1 index and the index decrements from right to left, as a result the first character H has -12 index.

negative indexing

Let us use negative indexing to fetch N, Y, and H characters.

>>> var[-1]

'N'

>>> var[-5]

'Y'

>>> var[-12]

'H'

>>> var[-13]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: string index out of range

Once again, if the index goes beyond the range, IndexError is encountered.

We can therefore use positive or negative index to retrieve a character from the string.

>>> var[0], var[-12]

('H', 'H')

>>> var[7], var[-5]

('Y', 'Y')

>>> var[11], var[-1]

('N', 'N')

In Python, string is an immutable object. The object is immutable if it cannot be modified in-place, once stored in a certain memory location. You can retrieve any character from the string with the help of its index, but you cannot replace it with another character. In our example, character Y is at index 7 in HELLO PYTHON. Try to replace Y with y and see what happens.

var="HELLO PYTHON"

var[7]="y"

print (var)

It will produce the following **output** −

Traceback (most recent call last):

File "C:\Users\users\example.py", line 2, in <module>

var[7]="y"

~~~^^^

TypeError: 'str' object does not support item assignment

The TypeError is because the string is immutable.

Python defines ":" as string slicing operator. It returns a substring from the original string. Its general usage is −

substr=var[x:y]

The ":" operator needs two integer operands (both of which may be omitted, as we shall see in subsequent examples). The first operand x is the index of the first character of the desired slice. The second operand y is the index of the character next to the last in the desired string. So var(x:y] separates characters from xth position to (y-1)th position from the original string.

var="HELLO PYTHON"

print ("var:",var)

print ("var[3:8]:", var[3:8])

It will produce the following **output** −

var: HELLO PYTHON

var[3:8]: LO PY

Negative indexes can also be used for slicing.

var="HELLO PYTHON"

print ("var:",var)

print ("var[3:8]:", var[3:8])

print ("var[-9:-4]:", var[-9:-4])

It will produce the following **output** −

var: HELLO PYTHON

var[3:8]: LO PY

var[-9:-4]: LO PY

Both the operands for Python's Slice operator are optional. The first operand defaults to zero, which means if we do not give the first operand, the slice starts of character at 0th index, i.e. the first character. It slices the leftmost substring up to "y-1" characters.

var="HELLO PYTHON"

print ("var:",var)

print ("var[0:5]:", var[0:5])

print ("var[:5]:", var[:5])

It will produce the following **output** −

var: HELLO PYTHON

var[0:5]: HELLO

var[:5]: HELLO

Similarly, y operand is also optional. By default, it is "-1", which means the string will be sliced from the xth position up to the end of string.

var="HELLO PYTHON"

print ("var:",var)

print ("var[6:12]:", var[6:12])

print ("var[6:]:", var[6:])

It will produce the following output −

var: HELLO PYTHON

var[6:12]: PYTHON

var[6:]: PYTHON

he "+" operator is well-known as an addition operator, returning the sum of two numbers. However, the "+" symbol acts as string **concatenation operator** in Python. It works with two string operands, and results in the concatenation of the two.

The characters of the string on the right of plus symbol are appended to the string on its left. Result of concatenation is a new string.

str1="Hello"

str2="World"

print ("String 1:",str1)

print ("String 2:",str2)

str3=str1+str2

print("String 3:",str3)

It will produce the following **output** −

String 1: Hello

String 2: World

String 3: HelloWorld

To insert a whitespace between the two, use a third empty string.

str1="Hello"

str2="World"

blank=" "

print ("String 1:",str1)

print ("String 2:",str2)

str3=str1+blank+str2

print("String 3:",str3)

It will produce the following **output** −

String 1: Hello

String 2: World

String 3: Hello World

Another symbol \*, which we normally use for multiplication of two numbers, can also be used with string operands. Here, \* acts as a repetition operator in Python. One of the operands must be an integer, and the second a string. The operator concatenates multiple copies of the string. For example −

>>> "Hello"\*3

'HelloHelloHello'

The integer operand is the number of copies of the string operand to be concatenated.

Both the string operators, (\*) the repetition operator and (+) the concatenation operator, can be used in a single expression. The "\*" operator has a higher precedence over the "+" operator.

str1="Hello"

str2="World"

print ("String 1:",str1)

print ("String 2:",str2)

str3=str1+str2\*3

print("String 3:",str3)

str4=(str1+str2)\*3

print ("String 4:", str4)

To form **str3** string, Python concatenates 3 copies of World first, and then appends the result to Hello

String 3: HelloWorldWorldWorld

In the second case, the strings str1 and str2 are inside parentheses, hence their concatenation takes place first. Its result is then replicated three times.

String 4: HelloWorldHelloWorldHelloWorld

Apart from + and \*, no other arithmetic operator symbols can be used with string operands.

|  |  |
| --- | --- |
| **Sr.No** | **Escape Sequence & Meaning** |
| 1 | **\<newline>**  Backslash and newline ignored |
| 2 | **\\**  Backslash (\) |
| 3 | **\'**  Single quote (') |
| 4 | **\"**  Double quote (") |
| 5 | **\a**  ASCII Bell (BEL) |
| 6 | **\b**  ASCII Backspace (BS) |
| 7 | **\f**  ASCII Formfeed (FF) |
| 8 | **\n**  ASCII Linefeed (LF) |
| 9 | **\r**  ASCII Carriage Return (CR) |
| 10 | **\t**  ASCII Horizontal Tab (TAB) |
| 11 | **\v**  ASCII Vertical Tab (VT) |
| 12 | **\ooo**  Character with octal value ooo |
| 13 | **\xhh**  Character with hex value hh |
| Method | **Description** |
| [capitalize()](https://www.w3schools.com/python/ref_string_capitalize.asp) | **Converts the first character to upper case** |
| [casefold()](https://www.w3schools.com/python/ref_string_casefold.asp) | **Converts string into lower case** |
| [center()](https://www.w3schools.com/python/ref_string_center.asp) | **Returns a centered string** |
| [count()](https://www.w3schools.com/python/ref_string_count.asp) | **Returns the number of times a specified value occurs in a string** |
| [encode()](https://www.w3schools.com/python/ref_string_encode.asp) | **Returns an encoded version of the string** |
| [endswith()](https://www.w3schools.com/python/ref_string_endswith.asp) | **Returns true if the string ends with the specified value** |
| [expandtabs()](https://www.w3schools.com/python/ref_string_expandtabs.asp) | **Sets the tab size of the string** |
| [find()](https://www.w3schools.com/python/ref_string_find.asp) | **Searches the string for a specified value and returns the position of where it was found** |
| [format()](https://www.w3schools.com/python/ref_string_format.asp) | **Formats specified values in a string** |
| format\_map() | **Formats specified values in a string** |
| [index()](https://www.w3schools.com/python/ref_string_index.asp) | **Searches the string for a specified value and returns the position of where it was found** |
| [isalnum()](https://www.w3schools.com/python/ref_string_isalnum.asp) | **Returns True if all characters in the string are alphanumeric** |
| [isalpha()](https://www.w3schools.com/python/ref_string_isalpha.asp) | **Returns True if all characters in the string are in the alphabet** |
| [isascii()](https://www.w3schools.com/python/ref_string_isascii.asp) | **Returns True if all characters in the string are ascii characters** |
| [isdecimal()](https://www.w3schools.com/python/ref_string_isdecimal.asp) | **Returns True if all characters in the string are decimals** |
| [isdigit()](https://www.w3schools.com/python/ref_string_isdigit.asp) | **Returns True if all characters in the string are digits** |
| [isidentifier()](https://www.w3schools.com/python/ref_string_isidentifier.asp) | **Returns True if the string is an identifier** |
| [islower()](https://www.w3schools.com/python/ref_string_islower.asp) | **Returns True if all characters in the string are lower case** |
| [isnumeric()](https://www.w3schools.com/python/ref_string_isnumeric.asp) | **Returns True if all characters in the string are numeric** |
| [isprintable()](https://www.w3schools.com/python/ref_string_isprintable.asp) | **Returns True if all characters in the string are printable** |
| [isspace()](https://www.w3schools.com/python/ref_string_isspace.asp) | **Returns True if all characters in the string are whitespaces** |
| [istitle()](https://www.w3schools.com/python/ref_string_istitle.asp) | **Returns True if the string follows the rules of a title** |
| [isupper()](https://www.w3schools.com/python/ref_string_isupper.asp) | **Returns True if all characters in the string are upper case** |
| [join()](https://www.w3schools.com/python/ref_string_join.asp) | **Converts the elements of an iterable into a string** |
| [ljust()](https://www.w3schools.com/python/ref_string_ljust.asp) | **Returns a left justified version of the string** |
| [lower()](https://www.w3schools.com/python/ref_string_lower.asp) | **Converts a string into lower case** |
| [lstrip()](https://www.w3schools.com/python/ref_string_lstrip.asp) | **Returns a left trim version of the string** |
| [maketrans()](https://www.w3schools.com/python/ref_string_maketrans.asp) | **Returns a translation table to be used in translations** |
| [partition()](https://www.w3schools.com/python/ref_string_partition.asp) | **Returns a tuple where the string is parted into three parts** |
| [replace()](https://www.w3schools.com/python/ref_string_replace.asp) | **Returns a string where a specified value is replaced with a specified value** |
| [rfind()](https://www.w3schools.com/python/ref_string_rfind.asp) | **Searches the string for a specified value and returns the last position of where it was found** |
| [rindex()](https://www.w3schools.com/python/ref_string_rindex.asp) | **Searches the string for a specified value and returns the last position of where it was found** |
| [rjust()](https://www.w3schools.com/python/ref_string_rjust.asp) | **Returns a right justified version of the string** |
| [rpartition()](https://www.w3schools.com/python/ref_string_rpartition.asp) | **Returns a tuple where the string is parted into three parts** |
| [rsplit()](https://www.w3schools.com/python/ref_string_rsplit.asp) | **Splits the string at the specified separator, and returns a list** |
| [rstrip()](https://www.w3schools.com/python/ref_string_rstrip.asp) | **Returns a right trim version of the string** |
| [split()](https://www.w3schools.com/python/ref_string_split.asp) | **Splits the string at the specified separator, and returns a list** |
| [splitlines()](https://www.w3schools.com/python/ref_string_splitlines.asp) | **Splits the string at line breaks and returns a list** |
| [startswith()](https://www.w3schools.com/python/ref_string_startswith.asp) | **Returns true if the string starts with the specified value** |
| [strip()](https://www.w3schools.com/python/ref_string_strip.asp) | **Returns a trimmed version of the string** |
| [swapcase()](https://www.w3schools.com/python/ref_string_swapcase.asp) | **Swaps cases, lower case becomes upper case and vice versa** |
| [title()](https://www.w3schools.com/python/ref_string_title.asp) | **Converts the first character of each word to upper case** |
| [translate()](https://www.w3schools.com/python/ref_string_translate.asp) | **Returns a translated string** |
| [upper()](https://www.w3schools.com/python/ref_string_upper.asp) | **Converts a string into upper case** |
| [zfill()](https://www.w3schools.com/python/ref_string_zfill.asp) | **Fills the string with a specified number of 0 values at the beginning** |

Formatting In string

String formatting allows you to create dynamic strings by combining variables and values. In this article, we will discuss about 5 ways to format a string.

You will learn different methods of string formatting with examples for better understanding. Let’s look at them now!

How to Format Strings in Python

There are five different ways to perform string formatting in Python

Formatting with % Operator.

Formatting with format() string method.

Formatting with string literals, called f-strings.

Formatting with String Template Class

Formatting with center() string method.

So we will see the entirety of the above-mentioned ways, and we will also focus on which string formatting strategy is the best.

1. How to Format String using % Operator

It is the oldest method of string formatting. Here we use the modulo % operator. The modulo % is also known as the “string-formatting operator”.

Python Format String Using the % Operator

In the expression “The mangy, scrawny stray dog %s gobbled down” % ‘hurriedly’, the %s placeholder within the string is replaced by the value ‘hurriedly’.

print("The mangy, scrawny stray dog %s gobbled down" %'hurriedly' +

"the grain-free, organic dog food.")

Output

The mangy, scrawny stray dog hurriedly gobbled downthe grain-free, organic dog food.

Injecting Multiple Strings using the modulo Operator

Here we are inserting multiple strings with the % operator.

x = 'looked'

print("Misha %s and %s around"%('walked',x))

Output

Misha walked and looked around

Precision Handling in Python using % operator

Floating-point numbers use the format %a.bf. Here, a would be the minimum number of digits to be present in the string; these might be padded with white space if the whole number doesn’t have this many digits.

Close to this, bf represents how many digits are to be displayed after the decimal point.

In this code, the string ‘The value of pi is: %5.4f’ contains a format specifier %5.4f. The %5.4f format specifier is used to format a floating-point number with a minimum width of 5 and a precision of 4 decimal places.

print('The value of pi is: %5.4f' %(3.141592))

Output

The value of pi is: 3.1416

Multiple format conversion types

In the given code, the formatting string Python is converted to Integer and floating point with %d,%f.

variable = 12

string = "Variable as integer = %d \n\

Variable as float = %f" %(variable, variable)

print (string)

Output

Variable as integer = 12

Variable as float = 12.000000

Note: To know more about %-formatting, refer to String Formatting in Python using %

2. How to Format String using format() Method

Format() method was introduced with Python3 for handling complex string formatting more efficiently.

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: ‘String here {} then also {}’.format(‘something1′,’something2’)

Formatting String Python using format() Method

This code is using {} as a placeholder and then we have called.format() method on the ‘equal’ to the placeholder.

print('We all are {}.'.format('equal'))

Output

We all are equal.

Index-based Insertion

In this code, curly braces {} with indices are used within the string ‘{2} {1} {0}’ to indicate the positions where the corresponding values will be placed.

print('{2} {1} {0}'.format('directions',

'the', 'Read'))

Output

Read the directions

Insert object by Assigning Keywords

In this code, curly braces {} with named placeholders ({a}, {b}, {c}) are used within the string ‘a: {a}, b: {b}, c: {c}’ to indicate the positions where the corresponding named arguments will be placed.

print('a: {a}, b: {b}, c: {c}'.format(a = 1,

b = 'Two',

c = 12.3))

Output

a: 1, b: Two, c: 12.3

Reuse the inserted objects

In this code, curly braces {} with named placeholders ({p}) are used within the string ‘The first {p} was alright, but the {p} {p} was tough.’ to indicate the positions where the corresponding named argument p will be placed.

print(

'The first {p} was alright, but the {p} {p} was tough.'.format(p='second'))

Output

The first second was alright, but the second second was tough.

Float Precision with the.format() Method

Syntax: {[index]:[width][.precision][type]}

The type can be used with format codes:

‘d’ for integers

‘f’ for floating-point numbers

‘b’ for binary numbers

‘o’ for octal numbers

‘x’ for octal hexadecimal numbers

‘s’ for string

‘e’ for floating-point in an exponent format

Example:

Both the codes are doing string formatting. The first String is formatted with ‘%’ and the second String is formatted with .format().

print('The valueof pi is: %1.5f' %3.141592)

print('The valueof pi is: {0:1.5f}'.format(3.141592))

Output

The valueof pi is: 3.14159

The valueof pi is: 3.14159

Note: To know more about str.format(), refer to format() function in Python

3. Understanding Python f-string

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-String in Python is to make string interpolation simpler.

To create an f-string in Python, 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.

String Formatting with F-Strings

In this code, the f-string f”My name is {name}.” is used to interpolate the value of the name variable into the string.

name = 'Ele'

print(f"My name is {name}.")

Output

My name is Ele.

This new formatting syntax is very powerful and easy. You can also insert arbitrary Python expressions and you can even do arithmetic operations in it.

Arithmetic operations using F-strings

In this code, the f-string f” He said his age is {2 \* (a + b)}.” is used to interpolate the result of the expression 2 \* (a + b) into the string.

a = 5

b = 10

print(f"He said his age is {2 \* (a + b)}.")

Output

He said his age is 30.

We can also use lambda expressions in f-string formatting.

Lambda Expressions using F-strings

In this code, an anonymous lambda function is defined using lambda x: x\*2. This lambda function takes an argument x and returns its double.

print(f"He said his age is {(lambda x: x\*2)(3)}")

Output

He said his age is 6

Float precision in the f-String Method

In this code, f-string formatting is used to interpolate the value of the num variable into the string.

Syntax: {value:{width}.{precision}}

num = 3.14159

print(f"The valueof pi is: {num:{1}.{5}}")

Output

The valueof pi is: 3.1416

Note: To know more about f-strings, refer to f-strings in Python

4. Python String Template Class

In the String module, Template Class allows us to create simplified syntax for output specification. The format uses placeholder names formed by $ with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing $$ creates a single escaped $:

Formatting String Python Using Template Class

This code imports the Template class from the string module. The Template class allows us to create a template string with placeholders that can be substituted with actual values. Here we are substituting the values n1 and n2 in place of n3 and n4 in the string n.

from string import Template

n1 = 'Hello'

n2 = 'GeeksforGeeks'

n = Template('$n3 ! This is $n4.')

# and pass the parameters into the

# template string.

print(n.substitute(n3=n1, n4=n2))

Output

Hello ! This is GeeksforGeeks.

Note: To know more about the String Template class, refer to String Template Class in Python

5. How to Format String using center() Method

The center() method is a built-in method in Python’s str class that returns a new string that is centered within a string of a specified width.

Formatting string using center() method

This code returns a new string padded with spaces on the left and right sides.

string = "GeeksForGeeks!"

width = 30

centered\_string = string.center(width)

print(centered\_string)

Output

GeeksForGeeks!

Python Format String: % vs. .format vs. f-string literal

f-strings are faster and better than both %-formatting and str.format(). f-strings expressions are evaluated at runtime, and we can also embed expressions inside f-string, using a very simple and easy syntax.

The expressions inside the braces are evaluated in runtime and then put together with the string part of the f-string and then the final string is returned.

Note: Use Template String if the string is a user-supplied string Else Use f-Strings if you are on Python 3.6+, and. format() method if you are not.

We have covered all 5 ways of string formatting in Python. There are many use cases and examples for each method. We also compared these methods to find which one is most efficient to use in real-life projects.

Similar Reads:

String Formatting in Python

How to use string formatters in Python

How to format a string using a dictionary in Python

Don't miss your chance to ride the wave of the data revolution! Every industry is scaling new heights by tapping into the power of data. Sharpen your skills and become a part of the hottest trend in the 21st century.

Dive into the future of technology - explore the Complete Machine Learning and Data Science Program by GeeksforGeeks and stay ahead of the curve

Regular Expressions

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

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

**Regex Module in Python**

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

**Example:** Importing re module in Python

* Python3

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

**How to Use RegEx in Python?**

You can use RegEx in Python after importing re module.

**Example:**

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

* Python3

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

**Output**

Start Index: 34

End Index: 40

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

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

**Metacharacters**

Metacharacters are the characters with special meaning.

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

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

Verbose

# Verbose in Python Regex

**Read**

Courses

Practice

Jobs

In this article, we will learn about **VERBOSE** flag of the **re package** and how to use it. **re.VERBOSE :** This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like \*?, (?: or (?P. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

|  |
| --- |
| # Without Using VERBOSE  regex\_email **=** re.compile(r'^([a-z0-9\_\.-]+)@([0-9a-z\.-]+)\.([a-z\.]{2, 6})$',                re.IGNORECASE)    # Using VERBOSE  regex\_email **=** re.compile(r"""              ^([a-z0-9\_\.-]+)              # local Part              @                             # single @ sign              ([0-9a-z\.-]+)                # Domain name              \.                            # single Dot .              ([a-z]{2,6})$                 # Top level Domain               """,re.VERBOSE | re.IGNORECASE) |

It’s passed as an argument to re.compile() i.e **re.compile(Regular Expression, re.VERBOSE)**. re.compile() returns a RegexObject which is then matched with the given string. Let’s consider an example where the user is asked to enter their Email ID and we have to validate it using RegEx. The format of an email is as follow:

* Personal details/local part like john123
* Single @
* Domain Name like gmail/yahoo etc
* Single Dot(.)
* Top Level Domain like .com/.org/.net

**Examples:**

**Input** : expectopatronum@gmail.com

**Output** : Valid

**Input** : avadakedavra@yahoo.com@

**Output** : Invalid

This is invalid because there is @ after the top level domain name.

module declaration

# Python Modules

**Read**

Courses

Practice

Video

Jobs

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

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

## What is Python Module

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

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

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

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

**Example:**

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

* Python3

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

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

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

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

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

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

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

import module

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

**Importing modules in Python Example**

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

0 seconds of 17 secondsVolume 0%

* Python3

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

**Output:**

12

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

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

### Import Specific Attributes from a Python module

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

* Python3

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

**Output:**

4.0  
720

## Import all Names

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

**Syntax:**

from module\_name import \*

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

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

* Python3

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

**Output**

4.0  
720

## Locating Python Modules

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

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

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

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

* Python3

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

**Output:**

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

## Renaming the Python Module

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

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

* Python3

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

**Output**

4.0

720

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

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

* Python3

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

**Output:**

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

We have covered Python Modules and it’s operations like create, import, etc. This article will give the overview about Python modules so that you can easily create and use modules in Python.

<https://www.programiz.com/python-programming/function>

Objects

An **Object** is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. Python is an object-oriented programming language that stresses objects i.e. it mainly emphasizes functions. Python Objects are basically an encapsulation of data variables and methods acting on that data into a single entity.

**Note:** For more information, [Python Classes and Objects](https://www.geeksforgeeks.org/python-classes-and-objects/)

**Understanding of Python Object**

For a better understanding of the concept of objects in Python. Let’s consider an example, many of you have played CLASH OF CLANS, So let’s assume base layout as the class which contains all the buildings, defenses, resources, etc. Based on these descriptions we make a village, here the village is the object in Python.

**Syntax:**

obj = MyClass()

print(obj.x)

**Instance** defining represent memory allocation necessary for storing the actual data of variables. Each time when you create an object of class a copy of each data variable defined in that class is created. In simple language, we can state that each object of a class has its own copy of data members defined in that class.

**Creating a Python Object**

**Working of the Program:**Audi = Cars()

* A block of memory is allocated on the heap. The size of memory allocated is decided by the attributes and methods available in that class(Cars).
* After the memory block is allocated, the special method [\_\_init\_\_](https://www.geeksforgeeks.org/__init__-in-python/)() is called internally. Initial data is stored in the variables through this method.
* The location of the allocated memory address of the instance is returned to the object(Cars).
* The memory location is passed to [self](https://www.geeksforgeeks.org/self-in-python-class/).
* Python3

|  |
| --- |
| **class** Cars:  **def** \_\_init\_\_(self, m, p):      self.model **=** m      self.price **=** p    Audi **=** Cars("R8", 100000)    print(Audi.model)  print(Audi.price) |

**Output:**

R8

100000

Exceptions

# **Python Exceptions**

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception. We will see what an exception is. Also, we will see the difference between a syntax error and an exception in this tutorial. Following that, we will learn about trying and except blocks and how to raise exceptions and make assertions. After that, we will see the Python exceptions list.

## What is an Exception?

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

### Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur. Consider the following scenario:

Backward Skip 10sPlay VideoForward Skip 10s

**Code**

1. #Python code after removing the syntax error
2. string = "Python Exceptions"
4. **for** s **in** string:
5. **if** (s != o:
6. **print**( s )

**Output:**

if (s != o:

^

SyntaxError: invalid syntax

The arrow in the output shows where the interpreter encountered a syntactic error. There was one unclosed bracket in this case. Close it and rerun the program:

**Code**

1. #Python code after removing the syntax error
2. string = "Python Exceptions"
4. **for** s **in** string:
5. **if** (s != o):
6. **print**( s )

**Output:**

2 string = "Python Exceptions"

4 for s in string:

----> 5 if (s != o):

6 print( s )

NameError: name 'o' is not defined

We encountered an exception error after executing this code. When syntactically valid Python code produces an error, this is the kind of error that arises. The output's last line specified the name of the exception error code encountered. Instead of displaying just "exception error", Python displays information about the sort of exception error that occurred. It was a NameError in this situation. Python includes several built-in exceptions. However, Python offers the facility to construct custom exceptions.

ADVERTISEMENT

ADVERTISEMENT

## Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

**Code**

1. # Python code to catch an exception and handle it using try and except code blocks
3. a = ["Python", "Exceptions", "try and except"]
4. **try**:
5. #looping through the elements of the array a, choosing a range that goes beyond the length of the array
6. **for** i **in** range( 4 ):
7. **print**( "The index and element from the array is", i, a[i] )
8. #if an error occurs in the try block, then except block will be executed by the Python interpreter
9. **except**:
10. **print** ("Index out of range")

**Output:**

The index and element from the array is 0 Python

The index and element from the array is 1 Exceptions

The index and element from the array is 2 try and except

Index out of range

The code blocks that potentially produce an error are inserted inside the try clause in the preceding example. The value of i greater than 2 attempts to access the list's item beyond its length, which is not present, resulting in an exception. The except clause then catches this exception and executes code without stopping it.

## How to Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

**Code**

1. #Python code to show how to raise an exception in Python
2. num = [3, 4, 5, 7]
3. **if** len(num) > 3:
4. **raise** Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )

**Output:**

1 num = [3, 4, 5, 7]

2 if len(num) > 3:

----> 3 raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )

Exception: Length of the given list must be less than or equal to 3 but is 4

The implementation stops and shows our exception in the output, providing indications as to what went incorrect.

## Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.

Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.

ADVERTISEMENT

ADVERTISEMENT

Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

### The assert Statement

Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

**The syntax for the assert clause is −**

1. **assert** Expressions[, Argument]

Python uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.

**Code**

1. #Python program to show how to use assert keyword
2. # defining a function
3. **def** square\_root( Number ):
4. **assert** ( Number < 0), "Give a positive integer"
5. **return** Number\*\*(1/2)
7. #Calling function and passing the values
8. **print**( square\_root( 36 ) )
9. **print**( square\_root( -36 ) )

**Output:**

7 #Calling function and passing the values

----> 8 print( square\_root( 36 ) )

9 print( square\_root( -36 ) )

Input In [23], in square\_root(Number)

3 def square\_root( Number ):

----> 4 assert ( Number < 0), "Give a positive integer"

5 return Number\*\*(1/2)

AssertionError: Give a positive integer

## Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

Here is an instance of a try clause with an else clause.

**Code**

1. # Python program to show how to use else clause with try and except clauses
3. # Defining a function which returns reciprocal of a number
4. **def** reciprocal( num1 ):
5. **try**:
6. reci = 1 / num1
7. **except** ZeroDivisionError:
8. **print**( "We cannot divide by zero" )
9. **else**:
10. **print** ( reci )
11. # Calling the function and passing values
12. reciprocal( 4 )
13. reciprocal( 0 )

**Output:**

0.25

We cannot divide by zero

## Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

Here is an example of finally keyword with try-except clauses:

**Code**

1. # Python code to show the use of finally clause
3. # Raising an exception in try block
4. **try**:
5. div = 4 // 0
6. **print**( div )
7. # this block will handle the exception raised
8. **except** ZeroDivisionError:
9. **print**( "Atepting to divide by zero" )
10. # this will always be executed no matter exception is raised or not
11. **finally**:
12. **print**( 'This is code of finally clause' )

**Output:**

Atepting to divide by zero

This is code of finally clause

## User-Defined Exceptions

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.

Here is an illustration of a RuntimeError. In this case, a class that derives from RuntimeError is produced. Once an exception is detected, we can use this to display additional detailed information.

We raise a user-defined exception in the try block and then handle the exception in the except block. An example of the class EmptyError is created using the variable var.

**Code**

1. **class** EmptyError( RuntimeError ):
2. **def** \_\_init\_\_(self, argument):
3. self.arguments = argument
4. Once the preceding **class** has been created, the following **is** how to **raise** an exception:
5. Code
6. var = " "
7. **try**:
8. **raise** EmptyError( "The variable is empty" )
9. **except** (EmptyError, var):
10. **print**( var.arguments )

**Output:**

2 try:

----> 3 raise EmptyError( "The variable is empty" )

4 except (EmptyError, var):

EmptyError: The variable is empty

## Python Exceptions List

Here is the complete list of Python in-built exceptions.

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Name of the Exception** | **Description of the Exception** |
| **1** | **Exception** | All exceptions of Python have a base class. |
| **2** | **StopIteration** | If the next() method returns null for an iterator, this exception is raised. |
| **3** | **SystemExit** | The sys.exit() procedure raises this value. |
| **4** | **StandardError** | Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions. |
| **5** | **ArithmeticError** | All mathematical computation errors belong to this base class. |
| **6** | **OverflowError** | This exception is raised when a computation surpasses the numeric data type's maximum limit. |
| **7** | **FloatingPointError** | If a floating-point operation fails, this exception is raised. |
| **8** | **ZeroDivisionError** | For all numeric data types, its value is raised whenever a number is attempted to be divided by zero. |
| **9** | **AssertionError** | If the Assert statement fails, this exception is raised. |
| **10** | **AttributeError** | This exception is raised if a variable reference or assigning a value fails. |
| **11** | **EOFError** | When the endpoint of the file is approached, and the interpreter didn't get any input value by raw\_input() or input() functions, this exception is raised. |
| **12** | **ImportError** | This exception is raised if using the import keyword to import a module fails. |
| **13** | **KeyboardInterrupt** | If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised. |
| **14** | **LookupError** | LookupErrorBase is the base class for all search errors. |
| **15** | **IndexError** | This exception is raised when the index attempted to be accessed is not found. |
| **16** | **KeyError** | When the given key is not found in the dictionary to be found in, this exception is raised. |
| **17** | **NameError** | This exception is raised when a variable isn't located in either local or global namespace. |
| **18** | **UnboundLocalError** | This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value. |
| **19** | **EnvironmentError** | All exceptions that arise beyond the Python environment have this base class. |
| **20** | **IOError** | If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised. |
| **22** | **SyntaxError** | This exception is raised whenever a syntax error occurs in our program. |
| **23** | **IndentationError** | This exception was raised when we made an improper indentation. |
| **24** | **SystemExit** | This exception is raised when the sys.exit() method is used to terminate the Python interpreter. The parser exits if the situation is not addressed within the code. |
| **25** | **TypeError** | This exception is raised whenever a data type-incompatible action or function is tried to be executed. |
| **26** | **ValueError** | This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values. |
| **27** | **RuntimeError** | This exception is raised when an error that occurred during the program's execution cannot be classified. |
| **28** | **NotImplementedError** | If an abstract function that the user must define in an inherited class is not defined, this exception is raised. |

## Summary

We learned about different methods to raise, catch, and handle Python exceptions after learning the distinction between syntax errors and exceptions. We learned about these clauses in this tutorial:

* We can throw an exception at any line of code using the raise keyword.
* Using the assert keyword, we may check to see if a specific condition is fulfilled and raise an exception if it is not.
* All statements are carried out in the try clause until an exception is found.
* The try clause's exception(s) are detected and handled using the except function.
* If no exceptions are thrown in the try code block, we can write code to be executed in the else code block.

Here is the syntax of try, except, else, and finally clauses.

Lambda Functions

**Python Lambda Functions** are anonymous functions means that the function is without a name. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in [Python](https://www.geeksforgeeks.org/python-programming-language/).

**Python Lambda Function Syntax**

***Syntax:****lambda arguments : expression*

* *This function can have any number of arguments but only one expression, which is evaluated and returned.*
* *One is free to use lambda functions wherever function objects are required.*
* *You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.*
* *It has various uses in particular fields of programming, besides other types of expressions in functions.*

**Python Lambda Function Example**

In the example, we defined a lambda function(**upper**) to convert a string to its upper case using [upper()](https://www.geeksforgeeks.org/python-string-upper/).

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

* Python3

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

**Output:**

GEEKSFORGEEKS

**Use of Lambda Function in Python**

Let’s see some of the practical uses of the Python lambda function.

**Condition Checking Using Python lambda function**

Here, the **‘format\_numric’** calls the lambda function, and the num is passed as a parameter to perform operations.

* Python3

|  |
| --- |
| format\_numeric **=** **lambda** num: f"{num:e}" **if** isinstance(num, int) **else** f"{num:,.2f}"    print("Int formatting:", format\_numeric(1000000))  **print**("float formatting:", format\_numeric(999999.789541235)) |

**Output:**

Int formatting: 1.000000e+06  
float formatting: 999,999.79

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

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

* Python3

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

**Output:**

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

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

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

UNIT III Classes

Creating classes, instance methods, Instance Variables, Closures, Generators, Iterators,Assert, Generator Expressions

# Python Classes and Objects

In the last tutorial, we learned about [Python OOP](https://www.programiz.com/python-programming/object-oriented-programming). We know that Python also supports the concept of objects and classes.

An object is simply a collection of data ([variables](https://www.programiz.com/python-programming/variables-constants-literals)) and methods ([functions](https://www.programiz.com/python-programming/function)). Similarly, a class is a blueprint for that object.

Before we learn about objects, let's first learn about classes in Python.

## Python Classes

A class is considered a blueprint of objects.

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc.

Based on these descriptions, we build the house; the house is the object.

Since many houses can be made from the same description, we can create many objects from a class.

## Define Python Class

We use the class [keyword](https://www.programiz.com/python-programming/keywords-identifier) to create a class in Python. For example,

class ClassName:

# class definition

Here, we have created a class named ClassName.

Let's see an example,

class Bike:

name = ""

gear = 0

Here,

1. Bike - the name of the class
2. name/gear - variables inside the class with default values "" and **0** respectively.

**Note**: The variables inside a class are called attributes.

## Python Objects

An object is called an instance of a class.

Suppose Bike is a class then we can create objects like bike1, bike2, etc from the class.

Here's the syntax to create an object.

objectName = ClassName()

Let's see an example,

# create class

class Bike:

name = ""

gear = 0

# create objects of class

bike1 = Bike()

Here, bike1 is the object of the class. Now, we can use this object to access the class attributes.

## Access Class Attributes Using Objects

We use the . notation to access the attributes of a class. For example,

# modify the name property

bike1.name = "Mountain Bike"

# access the gear property

bike1.gear

Here, we have used bike1.name and bike1.gear to change and access the value of name and gear attributes, respectively.

## Example 1: Python Class and Objects

# define a class

class Bike:

name = ""

gear = 0

# create object of class

bike1 = Bike()

# access attributes and assign new values

bike1.gear = 11

bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Name: Mountain Bike, Gears: 11

In the above example, we have defined the class named Bike with two attributes: name and gear.

We have also created an object bike1 of the class Bike.

Finally, we have accessed and modified the properties of an object using the . notation.

## Create Multiple Objects of Python Class

We can also create multiple objects from a single class. For example,

# define a class

class Employee:

# define a property

employee\_id = 0

# create two objects of the Employee class

employee1 = Employee()

employee2 = Employee()

# access property using employee1

employee1.employeeID = 1001

print(f"Employee ID: {employee1.employeeID}")

# access properties using employee2

employee2.employeeID = 1002

print(f"Employee ID: {employee2.employeeID}")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Employee ID: 1001

Employee ID: 1002

In the above example, we have created two objects employee1 and employee2 of the Employee class.

## Python Methods

We can also define a function inside a Python class. A Python function defined inside a class is called a **method**.

Let's see an example,

# create a class

class Room:

length = 0.0

breadth = 0.0

# method to calculate area

def calculate\_area(self):

print("Area of Room =", self.length \* self.breadth)

# create object of Room class

study\_room = Room()

# assign values to all the properties

study\_room.length = 42.5

study\_room.breadth = 30.8

# access method inside class

study\_room.calculate\_area()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Area of Room = 1309.0

In the above example, we have created a class named Room with:

1. **Attributes**: length and breadth
2. **Method**: calculate\_area()

Here, we have created an object named study\_room from the Room class.

We then used the object to assign values to attributes: length and breadth.

Notice that we have also used the object to call the method inside the class,

study\_room.calculate\_area()

Here, we have used the . notation to call the method. Finally, the statement inside the method is executed.

## Python Constructors

Earlier we assigned a default value to a class attribute,

class Bike:

name = ""

...

# create object

bike1 = Bike()

However, we can also initialize values using the constructors. For example,

class Bike:

# constructor function

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

self.name = name

bike1 = Bike()

Here, \_\_init\_\_() is the constructor function that is called whenever a new object of that class is instantiated.

The constructor above initializes the value of the name attribute.

We have used the self.name to refer to the name attribute of the bike1 object.

If we use a constructor to initialize values inside a class, we need to pass the corresponding value during the object creation of the class.

bike1 = Bike("Mountain Bike")

Here, "Mountain Bike" is passed to the name parameter of \_\_init\_\_().

<https://www.geeksforgeeks.org/python-classes-and-objects/>

instance methods

# Instance method in Python

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state. **Example:**

|  |
| --- |
| # Python program to demonstrate  # classes      **class** Person:        # init method or constructor  **def** \_\_init\_\_(self, name):          self.name **=** name        # Sample Method  **def** say\_hi(self):          print('Hello, my name is', self.name)    p **=** Person('Nikhil')  p.say\_hi() |

**Output:**

Hello, my name is Nikhil

**Note:** For more information, refer to [Python Classes and Objects](https://www.geeksforgeeks.org/python-classes-and-objects/)

#### Instance Method

Instance attributes are those attributes that are not shared by objects. Every object has its own copy of the instance attribute. For example, consider a class shapes that have many objects like circle, square, triangle, etc. having its own attributes and methods. An instance attribute refers to the properties of that particular object like edge of the triangle being 3, while the edge of the square can be 4. An instance method can access and even modify the value of attributes of an instance. It has one default parameter:-

* [self](https://www.geeksforgeeks.org/self-in-python-class/)**–** It is a keyword which points to the current passed instance. But it need not be passed every time while calling an instance method.

**Example:**

|  |
| --- |
| # Python program to demonstrate  # instance methods      **class** shape:        # Calling Constructor  **def** \_\_init\_\_(self, edge, color):          self.edge **=** edge          self.color **=** color        # Instance Method  **def** finEdges(self):  **return** self.edge        # Instance Method  **def** modifyEdges(self, newedge):          self.edge **=** newedge    # Driver Code  circle **=** shape(0, 'red')  square **=** shape(4, 'blue')    # Calling Instance Method  **print**("No. of edges for circle: "**+** str(circle.finEdges()))    # Calling Instance Method  square.modifyEdges(6)    **print**("No. of edges for square: "**+** str(square.finEdges())) |

**Output**

No. of edges for circle: 0

No. of edges for square: 6

Don't miss your chance to ride the wave of the data revolution! Every industry is scaling new heights by tapping into the power of data. Sharpen your skills and become a part of the hottest trend in the 21st century.

Dive into the future of technology - explore the [Complete Machine Learning and Data Science Program](https://www.geeksforgeeks.org/courses/data-science-live?utm_source=geeksforgeeks&utm_medium=article_bottom_text&utm_campaign=courses) by GeeksforGeeks and stay ahead of the curve.

Instance Variables

Instance Variables:Instance variables are unique to each instance of a class. They are defined within methods and are prefixed with the self keyword. These variables store data that is specific to an instance, making them essential for object-oriented programming (OOP) principles like encapsulation.

Closures

# Python Closures

Python closure is a nested [function](https://www.programiz.com/python-programming/function) that allows us to access [variables](https://www.programiz.com/python-programming/variables-constants-literals) of the outer function even after the outer function is closed.

Before we learn about closure, let's first revise the concept of nested functions in Python.

## Nested function in Python

In Python, we can create a function inside another function. This is known as a nested function. For example,

def greet(name):

# inner function

def display\_name():

print("Hi", name)

# call inner function

display\_name()

# call outer function

greet("John")

# Output: Hi John

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have defined the display\_name() function inside the greet() function.

Here, display\_name() is a nested function. The nested function works similar to the normal function. It executes when display\_name() is called inside the function greet().

## Python Closures

As we have already discussed, closure is a nested function that helps us access the outer function's variables even after the outer function is closed. For example,

def greet():

# variable defined outside the inner function

name = "John"

# return a nested anonymous function

return lambda: "Hi " + name

# call the outer function

message = greet()

# call the inner function

print(message())

# Output: Hi John

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have created a function named greet() that returns a nested [anonymous function](https://www.programiz.com/python-programming/anonymous-function).

Here, when we call the outer function,

message = greet()

The returned function is now assigned to the message variable.

At this point, the execution of the outer function is completed, so the name variable should be destroyed. However, when we call the anonymous function using

print(message())

we are able to access the name variable of the outer function.

It's possible because the nested function now acts as a closure that closes the outer scope variable within its [scope](https://www.programiz.com/python-programming/global-local-nonlocal-variables) even after the outer function is executed.

Let's see one more example to make this concept clear.

## Example: Print Odd Numbers using Python Closure

def calculate():

num = 1

def inner\_func():

nonlocal num

num += 2

return num

return inner\_func

# call the outer function

odd = calculate()

# call the inner function

print(odd())

print(odd())

print(odd())

# call the outer function again

odd2 = calculate()

print(odd2())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

3

5

7

3

In the above example,

odd = calculate()

This code executes the outer function calculate() and returns a closure to the odd number. T

That's why we can access the num variable of calculate() even after completing the outer function.

Again, when we call the outer function using

odd2 = calculate()

a new closure is returned. Hence, we get **3** again when we call odd2().

## When to use closures?

So what are closures good for?

Closures can be used to avoid global values and provide data hiding, and can be an elegant solution for simple cases with one or few methods.

However, for larger cases with multiple attributes and methods, a class implementation may be more appropriate.

def make\_multiplier\_of(n):

def multiplier(x):

return x \* n

return multiplier

# Multiplier of 3

times3 = make\_multiplier\_of(3)

# Multiplier of 5

times5 = make\_multiplier\_of(5)

# Output: 27

print(times3(9))

# Output: 15

print(times5(3))

# Output: 30

print(times5(times3(2)))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

[Python Decorators](https://www.programiz.com/python-programming/decorator) make extensive use of closures as well.

On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.

All function objects have a \_\_closure\_\_ attribute that returns a [tuple](https://www.programiz.com/python-programming/tuple) of cell objects if it is a closure function.

Generators

# Python Generators

In Python, a generator is a [function](https://www.programiz.com/python-programming/function) that returns an [iterator](https://www.programiz.com/python-programming/iterator) that produces a sequence of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

## Create Python Generator

In Python, similar to defining a [normal function](https://www.programiz.com/python-programming/function), we can define a generator function using the def [keyword](https://www.programiz.com/python-programming/keywords-identifier), but instead of the return statement we use the yield statement.

def generator\_name(arg):

# statements

yield something

Here, the yield keyword is used to produce a value from the generator.

When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

## Example: Python Generator

Here's an example of a generator function that produces a sequence of numbers,

def my\_generator(n):

# initialize counter

value = 0

# loop until counter is less than n

while value < n:

# produce the current value of the counter

yield value

# increment the counter

value += 1

# iterate over the generator object produced by my\_generator

for value in my\_generator(3):

# print each value produced by generator

print(value)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

2

In the above example, the my\_generator() generator function takes an integer n as an argument and produces a sequence of numbers from **0** to n-1 using [while loop](https://www.programiz.com/python-programming/while-loop).

The yield keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.

The for loop iterates over the generator object produced by my\_generator(), and the print statement prints each value produced by the generator.

We can also create a generator object from the generator function by calling the function like we would any other function as,

generator = my\_range(3)

print(next(generator)) # 0

print(next(generator)) # 1

print(next(generator)) # 2

**Note**: To learn more, visit [range()](https://www.programiz.com/python-programming/methods/built-in/range) and [for loop()](https://www.programiz.com/python-programming/for-loop).

## Python Generator Expression

In Python, a generator expression is a concise way to create a generator object.

It is similar to a [list comprehension](https://www.programiz.com/python-programming/list-comprehension), but instead of creating a [list](https://www.programiz.com/python-programming/list), it creates a generator object that can be iterated over to produce the values in the generator.

### Generator Expression Syntax

A generator expression has the following syntax,

(expression for item in iterable)

Here, expression is a value that will be returned for each item in the iterable.

The generator expression creates a generator object that produces the values of expression for each item in the iterable, one at a time, when iterated over.

## Example 2: Python Generator Expression

# create the generator object

squares\_generator = (i \* i for i in range(5))

# iterate over the generator and print the values

for i in squares\_generator:

print(i)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

4

9

16

Here, we have created the generator object that will produce the squares of the numbers **0** through **4** when iterated over.

And then, to iterate over the generator and get the values, we have used the for loop.

## Use of Python Generators

There are several reasons that make generators a powerful implementation.

### 1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of **2** using an iterator class.

class PowTwo:

def \_\_init\_\_(self, max=0):

self.n = 0

self.max = max

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.n > self.max:

raise StopIteration

result = 2 \*\* self.n

self.n += 1

return result

The above program was lengthy and confusing. Now, let's do the same using a generator function.

def PowTwoGen(max=0):

n = 0

while n < max:

yield 2 \*\* n

n += 1

Since generators keep track of details automatically, the implementation was concise and much cleaner.

### 2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

### 3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

def all\_even():

n = 0

while True:

yield n

n += 2

### 4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the [Fibonacci series](https://www.programiz.com/python-programming/examples/fibonacci-sequence). And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

def fibonacci\_numbers(nums):

x, y = 0, 1

for \_ in range(nums):

x, y = y, x+y

yield x

def square(nums):

for num in nums:

yield num\*\*2

print(sum(square(fibonacci\_numbers(10))))

# Output: 4895

[Run Code](https://www.programiz.com/python-programming/online-compiler)

This pipelining is efficient and easy to read (and yes, a lot cooler!).

Iterators

# Iterators in Python

**Read**

Courses

Practice

Jobs

An iterator in Python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The Python iterators object is initialized using the **iter()**method. It uses the **next()** method for iteration.

1. **\_\_iter\_\_():** The iter() method is called for the initialization of an iterator. This returns an iterator object
2. **\_\_next\_\_():**The next method returns the next value for the iterable. When we use a for loop to traverse any iterable object, internally it uses the iter() method to get an iterator object, which further uses the next() method to iterate over. This method raises a StopIteration to signal the end of the iteration.

## Python iter() Example

* Python3

|  |
| --- |
| string **=** "GFG"  ch\_iterator **=** iter(string)    print(next(ch\_iterator))  **print**(next(ch\_iterator))  print(next(ch\_iterator)) |

**Output :**

G

F

G

### Creating and looping over an iterator using iter() and next()

Below is a simple Python Iterator that creates an iterator type that iterates from 10 to a given limit. For example, if the limit is 15, then it prints 10 11 12 13 14 15. And if the limit is 5, then it prints nothing.

* Python3

|  |
| --- |
| # A simple Python program to demonstrate  # working of iterators using an example type  # that iterates from 10 to given value    # An iterable user defined type  **class** Test:        # Constructor  **def** \_\_init\_\_(self, limit):          self.limit **=** limit        # Creates iterator object      # Called when iteration is initialized  **def** \_\_iter\_\_(self):          self.x **=** 10  **return** self        # To move to next element. In Python 3,      # we should replace next with \_\_next\_\_  **def** \_\_next\_\_(self):            # Store current value ofx          x **=** self.x            # Stop iteration if limit is reached  **if** x > self.limit:  **raise** StopIteration            # Else increment and return old value          self.x **=** x **+** 1;  **return** x    # Prints numbers from 10 to 15  **for** i **in** Test(15):  **print**(i)    # Prints nothing  **for** i **in** Test(5):      print(i) |

**Output:**

10

11

12

13

14

15

### Iterating over built-in iterable using iter() method

In the following iterations, the iteration state and iterator variable is managed internally (we can’t see it) using an iterator object to traverse over the built-in iterable like [list](https://www.geeksforgeeks.org/python-lists/),[tuple](https://www.geeksforgeeks.org/python-tuples/), [dict](https://www.geeksforgeeks.org/python-dictionary/), etc.

* Python3

|  |
| --- |
| # Sample built-in iterators    # Iterating over a list  **print**("List Iteration")  l **=** ["geeks", "for", "geeks"]  **for** i **in** l:      print(i)    # Iterating over a tuple (immutable)  **print**("\nTuple Iteration")  t **=** ("geeks", "for", "geeks")  **for** i **in** t:      print(i)    # Iterating over a String  **print**("\nString Iteration")  s **=** "Geeks"  **for** i **in** s :      print(i)    # Iterating over dictionary  **print**("\nDictionary Iteration")  d **=** dict()  d['xyz'] **=** 123  d['abc'] **=** 345  **for** i **in** d :  **print**("%s  %d" **%**(i, d[i])) |

**Output:**

List Iteration

geeks

for

geeks

Tuple Iteration

geeks

for

geeks

String Iteration

G

e

e

k

s

Dictionary Iteration

xyz 123

abc 345

## Iterable vs Iterator

Python iterable and Python iterator are different. The main difference between them is, iterable in Python cannot save the state of the iteration, whereas in iterators the state of the current iteration gets saved.

**Note:**Every iterator is also an iterable, but not every iterable is an iterator in [Python](https://www.geeksforgeeks.org/python-programming-language/).   
**Read more** – [Difference between iterable and iterator](https://www.geeksforgeeks.org/python-difference-iterable-iterator/).

### Iterating on an Iterable

Iterating on each item of the iterable.

* Python3

|  |
| --- |
| tup **=** ('a', 'b', 'c', 'd', 'e')    **for** item **in** tup:      print(item) |

**Output:**

a

b

c

d

e

### Iterating on an iterator

* Python3

|  |
| --- |
| tup **=** ('a', 'b', 'c', 'd', 'e')    # creating an iterator from the tuple  tup\_iter **=** iter(tup)    **print**("Inside loop:")  # iterating on each item of the iterator object  **for** index, item **in** enumerate(tup\_iter):      print(item)        # break outside loop after iterating on 3 elements  **if** index **==** 2:  **break**    # we can print the remaining items to be iterated using next()  # thus, the state was saved  print("Outside loop:")  **print**(next(tup\_iter))  **print**(next(tup\_iter)) |

**Output:**

Inside loop:

a

b

c

Outside loop:

d

e

### Getting StopIteration Error while using iterator

Iterable in Python can be iterated over multiple times, but iterators raise StopIteration Error when all items are already iterated.

Here, we are trying to get the next element from the iterator after the completion of the for-loop. Since the iterator is already exhausted, it raises a StopIteration Exception. Whereas, using an iterable, we can iterate on multiple times using for loop or can get items using indexing.

* Python3

|  |
| --- |
| iterable **=** (1, 2, 3, 4)  iterator\_obj **=** iter(iterable)    **print**("Iterable loop 1:")  # iterating on iterable  **for** item **in** iterable:  **print**(item, end**=**",")    **print**("\nIterable Loop 2:")  **for** item **in** iterable:  **print**(item, end**=**",")    **print**("\nIterating on an iterator:")  # iterating on an iterator object multiple times  **for** item **in** iterator\_obj:      print(item, end**=**",")    **print**("\nIterator: Outside loop")  # this line will raise StopIteration Exception  # since all items are iterated in the previous for-loop  print(next(iterator\_obj)) |

**Output:**

Iterable loop 1:

1,2,3,4,

Iterable Loop 2:

1,2,3,4,

Iterating on an iterator:

1,2,3,4,

Iterator: Outside loop

Traceback (most recent call last):

File "scratch\_1.py", line 21, in <module>

print(next(iterator\_obj))

StopIteration

Don't miss your chance to ride the wave of the data revolution! Every industry is scaling new heights by tapping into the power of data. Sharpen your skills and become a part of the hottest trend in the 21st century

Generator Expressions

# Python | Generator Expressions

**Read**

Courses

Practice

Jobs

In **Python**, to create iterators, we can use both regular functions and generators. **Generators** are written just like a normal function but we use yield() instead of return() for returning a result. It is more powerful as a tool to implement iterators. It is easy and more convenient to implement because it offers the evaluation of elements on demand. Unlike regular functions which on encountering a return statement terminates entirely, generators use a yield statement in which the state of the function is saved from the last call and can be picked up or resumed the next time we call a generator function. Another great advantage of the generator over a list is that it takes much less memory.

In addition to that, two more functions \_next\_() and \_iter\_() make the generator function more compact and reliable. Example :

* Python3

|  |
| --- |
| # Python code to illustrate generator, yield() and next().  **def** generator():      t **=** 1  **print** ('First result is ',t)  **yield** t        t **+=** 1  **print** ('Second result is ',t)  **yield** t        t **+=** 1  **print**('Third result is ',t)  **yield** t    call **=** generator()  next(call)  next(call)  next(call) |

Output :

First result is 1  
Second result is 2  
Third result is 3

**Difference between Generator function and Normal function –**

* Once the function yields, the function is paused and the control is transferred to the caller.
* When the function terminates, StopIteration is raised automatically on further calls.
* Local variables and their states are remembered between successive calls.
* The generator function contains one or more yield statements instead of a return statement.
* As the methods like \_next\_() and \_iter\_() are implemented automatically, we can iterate through the items using next().

There are various other expressions that can be simply coded similar to list comprehensions but instead of brackets we use parenthesis. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expression allows creating a generator without a yield keyword. However, it doesn’t share the whole power of the generator created with a yield function. Example :

* Python3

|  |
| --- |
| # Python code to illustrate generator expression  generator **=** (num **\*\*** 2 **for** num **in** range(10))  **for** num **in** generator:      print(num) |

Output :

0  
1  
4  
9  
16  
25  
36  
49  
64  
81

We can also generate a list using generator expressions :

* Python3

|  |
| --- |
| string **=** 'geek'  li **=** list(string[i] **for** i **in** range(len(string)**-**1, **-**1, **-**1))  print(li) |

Output:

['k', 'e', 'e', 'g']