**PYTHON NOTES**

**UNIT - 1**

**Definition**

**Python** is a dynamic, high-level, free open source, and interpreted programming language. It supports object-oriented programming as well as procedural-oriented programming. In Python, we don’t need to declare the type of variable because it is a dynamically typed language. For example, x = 10 Here, x can be anything such as String, int, etc.

## Features in Python

### ****1. Free and Open Source****

Python language is freely available at the official website and you can download it from the given download link below click on the **Download Python** keyword. Download Python Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.

### ****2. Easy to code****

Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.

### 3. Easy to Read

As you will see, learning Python is quite simple. As was already established, Python’s syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.

### ****4. Object-Oriented Language****

One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

### ****5. GUI Programming Support****

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in Python. PyQt5 is the most popular option for creating graphical apps with Python.

### ****6. High-Level Language****

Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

### ****7. Large Community Support****

Python has gained popularity over the years. Our questions are constantly answered by the enormous StackOverflow community. These websites have already provided answers to many questions about Python, so Python users can consult them as needed.

### 8. Easy to Debug

Excellent information for mistake tracing. You will be able to quickly identify and correct the majority of your program’s issues once you understand how to interpret Python’s error traces. Simply by glancing at the code, you can determine what it is designed to perform.

### ****9. Python is a Portable language****

Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

### ****10. Python is an Integrated language****

Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

### ****11. Interpreted Language:****

Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called **bytecode**.

### ****12. Large Standard Library****

Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.

### ****13. Dynamically Typed Language****

Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don’t need to specify the type of variable.

### ****14. Frontend and backend development****

With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it’s extensively used for this work cause of its frameworks like Django and Flask.

## Importance in Python

**1. Data Science**

Python is the preferred programming language of most data scientists. Be it IT ops, software development or marketing, currently every job makes use of data and depends on it to drive their operations. With the release of **‘Numpy‘** and **‘Pandas’**, Python rose to prominence in the world of data. Python also handles statistical, tabular and matrix data and also visualizes it with libraries like **‘Matplotlib’** and **‘Seaborn‘.**  
Moreover, in the data science arena, python job postings outnumber all others. This indicates the fact that the skills you gain with Python will directly transfer to developing your analytic skills.

**2. Easy to Learn**

**Python is an easy language to master.** This is chiefly because of its resemblance to the English language. Python’s syntax is characterized by very few rules and special cases. It’s safe to say that in Python the focus is on what you want to do with the code, not on language intricacies. Anybody can master Python easily. With practice, newbies can build a basic game in mere days using python. Another attractive aspect of this programming language is its efficiency and readability.

**3. Cross-Platform and Open Source**

It’s been more than 20 years since this language has been running cross-platform and open source. Be it **Linux, Windows** or **MacOS**, Python code works on every platform. Another remarkable thing about Python is that it’s supported by decades of bug-squashing and kink-straightening which ensures that its code works as intended whenever the user runs it.

**4. Versatile Language and Platform**

Python remains very relevant today as it can be used in any operations scenario or software development, be it in managing local and cloud infrastructure, working against a [SQL database](https://www.geeksforgeeks.org/sql-tutorial/), developing a custom function for [Hive](https://www.geeksforgeeks.org/hive-drop-table/) & [Pig,](https://www.geeksforgeeks.org/introduction-to-apache-pig/) supporting object-oriented design or even developing a small tool for the user.

**5. Vast Libraries**

Python is supported by **PyPI** which has *85,000+* python scripts and modules accessible to the user. These modules provide pre-packaged functionality available to the users in their local Python environment. **It can solve diverse problems such as executing advanced data analytics like developing REStful web services or sentiment analysis and establishing computer vision.**

**6. Flexibility**

Python has several powerful applications integrated with other programming languages. Details about these are given as follows:

* [.Net](https://www.geeksforgeeks.org/introduction-to-net-framework/) and [C#](https://www.geeksforgeeks.org/csharp-programming-language/) compatible: IronPython
* A version with [C](https://www.geeksforgeeks.org/c-programming-language/): CPython
* Python combined with[Ruby](https://www.geeksforgeeks.org/ruby-tutorial/): RubyPython
* Python integrated with [Java](https://www.geeksforgeeks.org/java/): [Jython](https://www.geeksforgeeks.org/jython-introduction-and-installation/" \t "_blank)
* Python written with Objective C toolkits: PyObjc

**7. Scripting and Automation**

What most people don’t know about Python is that it can be used as a scripting language. In scripting, the code is written in script form and gets executed. So the code is read and interpreted by the machine and errors are checked during runtime. After the code is checked, it can be used many times. It is also possible to automate specific tasks in a problem by automation.

**8. Artificial Intelligence**

[Artificial Intelligence (AI)](https://www.geeksforgeeks.org/artificial-intelligence-an-introduction/) is a pivotal domain where Python shines. Python’s libraries like [TensorFlow](https://www.geeksforgeeks.org/introduction-to-tensorflow/) and[Keras](https://www.geeksforgeeks.org/python-keras-keras-utils-to_categorical/) enable the implementation of complex [machine learning](https://www.geeksforgeeks.org/machine-learning/) and [deep learning](https://www.geeksforgeeks.org/deep-learning-tutorial/)algorithms. These libraries are widely used in AI applications such as[natural language processing (NLP)](https://www.geeksforgeeks.org/natural-language-processing-nlp-tutorial/), where Python’s NLTK and spaCy excel in tasks like sentiment analysis and language translation. In computer vision, Python’s [OpenCV](https://www.geeksforgeeks.org/opencv-overview/) library is a go-to tool for tasks like image recognition and object detection.

**9. Computer Graphics**

Python can be employed in small, large, online or offline projects. It is used to develop GUI and desktop applications. It’s **‘[Tkinter](https://www.geeksforgeeks.org/python-gui-tkinter/" \t "_blank)’** library enables simple and rapid application development. This programming language is also used in game development where the logic is written using a module *‘[pygame](https://www.geeksforgeeks.org/pygame-tutorial/" \t "_blank)’* which can also run on android devices.

**10. Testing Framework**

This language is an excellent tool for validating the products or ideas for established enterprises. Python has numerous built-in testing frameworks that deal with debugging and rapid workflows. Its tools and modules such as ***Selenium*** and ***Splinter*** work to make things easier. Python also supports cross-platform and cross-browser testing with frameworks like *PyTest* and *Robot* framework.

**11. Web Development**

**Python has a variety of framework support website or web-development**. Python has an array of frameworks for developing websites. Popular frameworks such as *Django, Flask,* and ***Pylons*** are characterized by faster and stable code; this is because they are written in Python. Using Python, the users can perform web scraping which means fetching details from other websites.

## What is Python Script?

Python is a well-known high-level programming language. The Python script is a file containing Python-written code. The file containing Python script has the extension ‘.py’ or can also have the extension ‘.**pyw**’ if it is being run on a Windows 10 machine. To run a Python script, we need a Python interpreter installed on the device.

**Methods to Run a Script in Python**

There are various methods to Run a Python script, we will go through some generally used methods for running a Python script:

* Interactive Mode
* Command Line
* Text Editor (VS Code)
* IDE (PyCharm)

## What are Data Types in Python ?

A data type in programming is like a label that tells the computer what kind of information is stored in a variable. It helps the computer understand how to handle and process that data, whether it's numbers, text, or other types of information.The following are the standard or built-in data types in Python:

* **Numeric**
* **Sequence Type**
* **Boolean**
* **Set**
* **Dictionary**



**Note** – **type() function** is used to **determine the data type** of a variable.

**(1) Numeric Data Types in Python**

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an **integer**, a **floating number**, or even a **complex number**. These values are defined as Python int, Python float, and Python complex classes in Python.

* **Integers**– This value is represented by **int class**. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
* **Float**– This value is represented by the **float class**. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
* **Complex Numbers** – A complex number is represented by a **complex class**. It is specified as *(real part) + (imaginary part) j*. For example – 2+3j

**Code:**

a **=** 5

**print**("Type of a: ", type(a))

b **=** 5.0

**print**("\nType of b: ", type(b))

c **=** 2 **+** 4j

print("\nType of c: ", type(c))

print(" c is a complex number", isinstance(1+3j,complex))

**Output:**

Type of a: <class 'int'>  
Type of b: <class 'float'>  
Type of c: <class 'complex'>

c is complex number: True

## (2) Sequence Data Type in Python

The sequence Data Type in Python is the ordered collection of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence types in Python –

(a) Python String

(b) Python List

(c) Python Tuple

### (a) String Data Type ->

[Strings](https://www.geeksforgeeks.org/python-strings/) in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In Python there is no character data type, a character is a string of length one. It is represented by str class.

#### **Creating String**

Strings in Python can be created using single quotes, double quotes, or even triple quotes.

**Example:** This Python code showcases various string creation methods. It uses single quotes, double quotes, and triple quotes to create strings with different content and includes a multiline string. The code also demonstrates printing the strings and checking their data types.

**Code:**

|  |
| --- |
| String1 **=** 'Welcome to the Geeks World'  print("String with the use of Single Quotes: ")  **print**(String1)  String1 **=** "I'm a Geek"  **print**("\nString with the use of Double Quotes: ")  print(String1)  **print**(type(String1))  String1 **=** '''I'm a Geek and I live in a world of "Geeks"'''  print("\nString with the use of Triple Quotes: ")  **print**(String1)  **print**(type(String1))  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  
<class 'str'>  
String with the use of Triple Quotes:   
I'm a Geek and I live in a world of "Geeks"  
<class 'str'>  
Creating a multiline String:   
Geeks   
 For   
 Life

**Accessing elements of String**

In Python programming, individual characters of a String can be accessed by using the method of Indexing. Negative 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.

**Code:**

|  |
| --- |
| String1 **=** "GeeksForGeeks"  **print**("Initial String: ")  print(String1)  **print**("\nFirst character of String is: ")  **print**(String1[0])  print("\nLast character of String is: ")  print(String1[**-**1]) |

**Output:**

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

**(b) List Data Type ->**

[Lists](https://www.geeksforgeeks.org/python-list/) are just like arrays, declared in other languages which is an **ordered collection of data**. It is very flexible as the items in a list do not need to be of the same type.

**Creating a List in Python**

Lists in Python can be created by just placing the sequence inside the **square brackets[ ].**

**Example:** This Python code demonstrates list creation and manipulation. It starts with an empty list and prints it. It creates a list containing a single string element and prints it. It creates a list with multiple string elements and prints selected elements from the list. It creates a multi-dimensional list (a list of lists) and prints it. The code showcases various ways to work with lists, including single and multi-dimensional lists.

**Code:**

|  |
| --- |
| List **=** []  **print**("Initial blank List: ")  **print**(List)  List **=** ['GeeksForGeeks']  print("\nList with the use of String: ")  print(List)  List **=** ["Geeks", "For", "Geeks"]  print("\nList containing multiple values: ")  print(List[0])  print(List[2])  List **=** [['Geeks', 'For'], ['Geeks']]  print("\nMulti-Dimensional List: ")  print(List) |

**Output:**

Initial blank List:   
[]  
List with the use of String:   
['GeeksForGeeks']  
List containing multiple values:   
Geeks  
Geeks  
Multi-Dimensional List:   
[['Geeks', 'For'], ['Geeks']]

**Python Access List Items**

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

**Code:**

|  |
| --- |
| List **=** ["Geeks", "For", "Geeks"]  **print**("Accessing element from the list")  **print**(List[0])  **print**(List[2])  print("Accessing element using negative indexing")  **print**(List[**-**1])  print(List[**-**3]) |

**Output:**

Accessing element from the list  
Geeks  
Geeks  
Accessing element using negative indexing  
Geeks  
Geeks

### (c) Tuple Data Type ->

Just like a list, a tuple is also an **ordered collection** of Python objects. The only difference between a tuple and a list is that tuples are **immutable** i.e. **tuples cannot be modified after it is created**.It is represented by a tuple class.

#### **Creating a Tuple in Python**

In Python, tuples are created by placing a sequence of values separated by a ‘comma’ **with or without the use of parentheses** for grouping the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.). **Note:** 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.

**Example:** This Python code demonstrates different methods of creating and working with tuples. It starts with an empty tuple and prints it. It creates a tuple containing string elements and prints it. It converts a list into a tuple and prints the result. It creates a tuple from a string using the **tuple()**function. It forms a tuple with nested tuples and displays the result.

**Code:**

|  |
| --- |
| Tuple1 **=** ()  **print**("Initial empty Tuple: ")  **print**(Tuple1)  Tuple1 **=** ('Geeks', 'For')  print("\nTuple with the use of String: ")  print(Tuple1)  list1 **=** [1, 2, 4, 5, 6]  print("\nTuple using List: ")  **print**(tuple(list1))  Tuple1 **=** tuple('Geeks')  print("\nTuple with the use of function: ")  **print**(Tuple1)  Tuple1 **=** (0, 1, 2, 3)  Tuple2 **=** ('python', 'geek')  Tuple3 **=** (Tuple1, Tuple2)  print("\nTuple with nested tuples: ")  print(Tuple3) |

**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')  
Tuple with nested tuples:   
((0, 1, 2, 3), ('python', 'geek'))

**Note**– The creation of a Python tuple **without the use of parentheses** is known as **Tuple Packing.**

#### Access Tuple Items

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

The code creates a tuple named ‘**tuple1′** with five elements:**1, 2, 3, 4, and 5**. Then it prints the first, last, and third last elements of the tuple using indexing.

**Code:**

|  |
| --- |
| tuple1 **=** tuple([1, 2, 3, 4, 5])  print("First element of tuple")  **print**(tuple1[0])  **print**("\nLast element of tuple")  print(tuple1[**-**1])  print("\nThird last element of tuple")  **print**(tuple1[**-**3]) |

**Output:**

First element of tuple  
1  
Last element of tuple  
5  
Third last element of tuple  
3

**(3) Boolean Data Type in Python**

Data type with one of the two built-in values, **True** or **False**. Boolean objects that are equal to True are **truthy** (true), and those equal to False are **falsy** (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by the class bool.

**Note**– True and False with **capital ‘T’ and ‘F’** are **valid booleans** otherwise python will throw an error.

**Example:** The first two lines will print the type of the boolean values True and False, which is **<class ‘bool’>.**The third line will cause an error, because true is not a valid keyword in Python. Python is case-sensitive, which means it distinguishes between uppercase and lowercase letters. You need to capitalize the first letter of true to make it a boolean value.

**Code:**

|  |
| --- |
| print(type(True))  print(type(False))    **print**(type(true)) |

**Output:**

<class 'bool'>  
<class 'bool'>

Traceback (most recent call last):  
 File "/home/7e8862763fb66153d70824099d4f5fb7.py", line 8, in   
 print(type(true))  
NameError: name 'true' is not defined

**(4) Set Data Type in Python**

In Python, a [Set](https://www.geeksforgeeks.org/python-sets/) 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.

**Create a Set in Python**

Sets can be created by using the built-in **set() function** with an iterable object or a sequence by placing the sequence **inside curly braces**, separated by a**‘comma’.** The type of elements in a set need not be the same, various **mixed-up data type** values can also be passed to the set.

**Code:**

|  |
| --- |
| set1 **=** set()  print("Initial blank Set: ")  **print**(set1)  set1 **=** set("GeeksForGeeks")  print("\nSet with the use of String: ")  **print**(set1)  set1 **=** set(["Geeks", "For", "Geeks"])  **print**("\nSet with the use of List: ")  **print**(set1)  set1 **=** set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])  **print**("\nSet with the use of Mixed Values")  print(set1) |

**Output:**

Initial blank Set:   
set()  
Set with the use of String:   
{'F', 'o', 'G', 's', 'r', 'k', 'e'}  
Set with the use of List:   
{'Geeks', 'For'}  
Set with the use of Mixed Values  
{1, 2, 4, 6, 'Geeks', 'For'}

**Access Set Items**

Set items **cannot be accessed** by referring to an **index**, since sets are **unordered** the items have 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.

**Code:**

|  |
| --- |
| set1 **=** set(["Geeks", "For", "Geeks"])  **print**("\nInitial set")  print(set1)  print("\nElements of set: ")  **for** i **in** set1:      print(i, end**=**" ")  print("Geeks" **in** set1) |

**Output:**

Initial set:   
{'Geeks', 'For'}  
Elements of set:   
Geeks For   
True

**(5) Dictionary Data Type in Python**

A dictionary in Python is an **unordered collection** of data values, used to store data values like a map, unlike other Data Types that hold only a single value as an element, a Dictionary **holds a key: value pair**. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon : , whereas each key is separated by a ‘comma’.

**Create a Dictionary in Python**

In Python, a Dictionary can be created by placing a sequence of elements within **curly { } braces**, separated by ‘comma’. **Values in a dictionary can be of any datatype and can be duplicated, whereas keys can’t be repeated and must be immutable**. The dictionary can also be created by the built-in function **dict().** An empty dictionary can be created by just placing it in curly braces{}. **Note**– Dictionary **keys** are **case sensitive**, the same name but different cases of Key will be treated distinctly.

**Example:** This code creates and prints a variety of dictionaries. The first dictionary is empty. The second dictionary has integer keys and string values. The third dictionary has mixed keys, with one string key and one integer key. The fourth dictionary is created using the **dict()** function, and the fifth dictionary is created using the **[(key, value)]**syntax

**Code:**

|  |
| --- |
| Dict **=** {}  **print**("Empty Dictionary: ")  print(Dict)  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)  Dict **=** dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})  **print**("\nDictionary with the use of dict(): ")  print(Dict)  Dict **=** dict([(1, 'Geeks'), (2, 'For')])  print("\nDictionary with each item as a pair: ")  print(Dict) |

**Output:**

Empty Dictionary:   
{}  
Dictionary with the use of Integer Keys:   
{1: 'Geeks', 2: 'For', 3: 'Geeks'}  
Dictionary with the use of Mixed Keys:   
{1: [1, 2, 3, 4], 'Name': 'Geeks'}  
Dictionary with the use of dict():   
{1: 'Geeks', 2: 'For', 3: 'Geeks'}  
Dictionary with each item as a pair:   
{1: 'Geeks', 2: 'For'}

**Accessing Key-value in Dictionary**

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a method called**get()** that will also help in **accessing the element** from a dictionary.

**Example:** The code in Python is used to access elements in a dictionary. Here’s what it does, It creates a dictionary Dict with keys and values as {**1: ‘Geeks’, ‘name’: ‘For’, 3: ‘Geeks’}**. It prints the value of the element with the key **‘name’**, which is **‘For’**. It prints the value of the element with the key 3, which is **‘Geeks’**.

**Code:**

|  |
| --- |
| Dict **=** {1: 'Geeks', 'name': 'For', 3: 'Geeks'}  **print**("Accessing a element using key:")  print(Dict['name'])  print("Accessing a element using get:")  **print**(Dict.get(3)) |

**Output:**

Accessing a element using key:  
For  
Accessing a element using get:  
Geeks

# (1) PYTHON LISTS -

The list is a sequence data type which is used to store the collection of data.

Lists need not be homogeneous always which makes it the most powerful tool in Python. 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

### Example :  Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

|  |
| --- |
| # Creating a List with  # the use of Numbers  # (Having duplicate values)  List **=** [1, 2, 4, 4, 3, 3, 3, 6, 5]  print("\nList with the use of Numbers: ")  print(List)    # Creating a List with  # mixed type of values  # (Having numbers and strings)  List **=** [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']  print("\nList with the use of Mixed Values: ")  **print**(List) |

**Output**

List with the use of Numbers:

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

List with the use of Mixed Values:

[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

**Example : Accessing elements from a multi-dimensional list**

|  |
| --- |
| # Creating a Multi-Dimensional List  # (By Nesting a list inside a List)  List **=** [['Geeks', 'For'], ['Geeks']]    # accessing an element from the  # Multi-Dimensional List using  # index number  print("Accessing a element from a Multi-Dimensional list")  print(List[0][1])  print(List[1][0]) |

**Output**

Accessing a element from a Multi-Dimensional list

For

Geeks

## Getting the size of Python list

Python **len( )** is used to get the length of the list.

|  |
| --- |
| # 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:**

|  |
| --- |
| # 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:**

|  |
| --- |
| # 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]

## 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.

|  |
| --- |
| # 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']]

### 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).

|  |
| --- |
| # 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]

### 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.*

|  |
| --- |
| # 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']

## Reversing a List

### Method 1:  A list can be reversed by using the [reverse() method in Python](https://www.geeksforgeeks.org/python-list-reverse/).

|  |
| --- |
| # Reversing a list  mylist **=** [1, 2, 3, 4, 5, 'Geek', 'Python']  mylist.reverse()  print(mylist) |

**Output**

['Python', 'Geek', 5, 4, 3, 2, 1]

### Method 2: Using the [reversed()](https://www.geeksforgeeks.org/python-reversed-function/) function:

The reversed() function returns a reverse iterator, which can be converted to a list using the list() function.

|  |
| --- |
| my\_list **=** [1, 2, 3, 4, 5]  reversed\_list **=** list(reversed(my\_list))  print(reversed\_list) |

**Output**

[5, 4, 3, 2, 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/) function but 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, the iterator is used. The remove() method removes the specified item.

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

**Example 1:**

|  |
| --- |
| # 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:**

|  |
| --- |
| # 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]

### Method 2: Using pop() method

[pop() function](https://www.geeksforgeeks.org/python-list-pop/) can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the pop() method.

|  |
| --- |
| List **=** [1, 2, 3, 4, 5]    # Removing element from the  # List using the pop() method  List.pop()  print("\nList after popping an element: ")  print(List)    # Removing element at a  # specific location from the  # List using the pop() method  List.pop(2)  **print**("\nList after popping a specific element: ")  print(List) |

**Output**

List after popping an element:

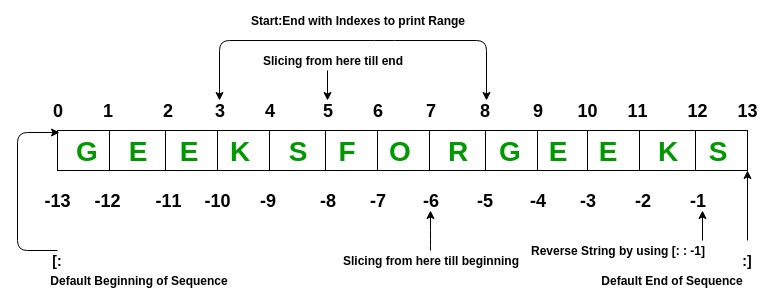
[1, 2, 3, 4]

List after popping a specific element:

[1, 2, 4]

## 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.

**Pr = [2, 3, 5, 7, 11, 13]**

**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.

|  |
| --- |
| # 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']

### Negative index List slicing

|  |
| --- |
| # Creating a List  List **=** ['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']  **print**("Initial List: ")  **print**(List)    # Print elements from beginning  # to a pre-defined point using Slice  Sliced\_List **=** List[:**-**6]  print("\nElements sliced till 6th element from last: ")  print(Sliced\_List)    # Print elements of a range  # using negative index List slicing  Sliced\_List **=** List[**-**6:**-**1]  **print**("\nElements sliced from index -6 to -1")  print(Sliced\_List)    # Printing elements in reverse  # using Slice operation  Sliced\_List **=** List[::**-**1]  print("\nPrinting List in reverse: ")  print(Sliced\_List) |

**Output**

Initial List:

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

Elements sliced till 6th element from last:

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

Elements sliced from index -6 to -1

['R', 'G', 'E', 'E', 'K']

Printing List in reverse:

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

## LIST METHODS

| **Methods** | **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. |

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. |

## 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 ]**

***Parameter:***

* **expression**: Represents the operation you want to execute on every item within the iterable.
* **element**: The term “variable” refers to each value taken from the iterable.
* **iterable**: specify the sequence of elements you want to iterate through.(e.g., a list, tuple, or string).
* **condition**: (Optional) A filter helps decide whether or not an element should be added to the new list.
* ***Return:****The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.*

*Python List comprehension provides a much more short syntax for creating a new list based on the values of an existing list.*

|  |
| --- |
| numbers **=** [1, 2, 3, 4, 5]  squared **=** [x **\*\*** 2 **for** x **in** numbers]  print(squared) |

**Output**

[1, 4, 9, 16, 25]

|  |
| --- |
| # 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:

|  |
| --- |
| # 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]

|  |
| --- |
| # Using list comprehension to iterate through loop  List **=** [character **for** character **in** [1, 2, 3]]    # Displaying list  print(List) |

**Output**

[1, 2, 3]

|  |
| --- |
| list **=** [i **for** i **in** range(11) **if** i **%** 2 **==** 0]  print(list) |

**Output**

[0, 2, 4, 6, 8, 10]

|  |
| --- |
| matrix **=** [[j **for** j **in** range(3)] **for** i **in** range(3)]    **print**(matrix) |

**Output**

[[0, 1, 2], [0, 1, 2], [0, 1, 2]]

|  |
| --- |
| # Reverse each string in tuple  List **=** [string[::**-**1] **for** string **in** ('Geeks', 'for', 'Geeks')]    # Display list  **print**(List) |

**Output**

['skeeG', 'rof', 'skeeG']

|  |
| --- |
| list **=** ["Even number" **if** i **%** 2 **==** 0  **else** "Odd number" **for** i **in** range(8)]  print(list) |

**Output**

['Even number', 'Odd number', 'Even number', 'Odd number', 'Even number', 'Odd number', 'Even number', 'Odd number']

|  |
| --- |
| numbers **=** [i**\***10 **for** i **in** range(1, 6)]    print(numbers) |

**Output**

[10, 20, 30, 40, 50]

# (2) TUPLES IN PYTHON -

Python Tuple is a **collection of objects** separated by commas. In some ways, a tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python **tuple is immutable**, unlike the Python list which is mutable.

## ****Creating Python Tuples****

There are various ways by which you can create a tuple in Python. They are as follows:

* Using round brackets
* With one item
* Tuple Constructor

### Create Tuples using Round Brackets ()

To create a tuple we will use () operators.

|  |
| --- |
| var **=** ("Geeks", "for", "Geeks")  print(var) |

**Output:**

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

### Create a Tuple With One Item

|  |
| --- |
| values : tuple[int | str, ...] **=** (1,2,4,"Geek")  print(values) |

**Output:**

Here, in the above snippet we are considering a variable called values which holds a tuple that consists of either int or str, the ‘…’ means that the tuple will hold more than one int or str.

(1, 2, 4, 'Geek')

**Note:** In case your generating a tuple with a single element, make sure to add a comma after the element. Let us see an example of the same.

|  |
| --- |
| mytuple **=** ("Geeks",)  **print**(type(mytuple))    #NOT a tuple  mytuple **=** ("Geeks")  print(type(mytuple)) |

**Output:**

<class 'tuple'>

<class 'str'>

### Tuple Constructor in Python

To create a tuple with a Tuple constructor, we will pass the elements as its parameters.

|  |
| --- |
| tuple\_constructor **=** tuple(("dsa", "developement", "deep learning"))  print(tuple\_constructor) |

**Output :**

('dsa', 'developement', 'deep learning')

## What is Immutable in Tuples?

Tuples in Python are similar to Python lists but not entirely. Tuples are immutable and ordered and allow duplicate values. Some Characteristics of Tuples in Python.

* We can find items in a tuple since finding any item does not make changes in the tuple.
* One cannot add items to a tuple once it is created.
* Tuples cannot be appended or extended.
* We cannot remove items from a tuple once it is created.

Let us see this with an example.

|  |
| --- |
| mytuple **=** (1, 2, 3, 4, 5)  # tuples are indexed  **print**(mytuple[1])  **print**(mytuple[4])  # tuples contain duplicate elements  mytuple **=** (1, 2, 3, 4, 2, 3)  **print**(mytuple)  # adding an element  mytuple[1] **=** 100  print(mytuple) |

**Output:**

Python tuples are ordered and we can access their elements using their index values. They are also immutable, i.e., we cannot add, remove and change the elements once declared in the tuple, so when we tried to add an element at index 1, it generated the error.

2

5

(1, 2, 3, 4, 2, 3)

Traceback (most recent call last):

File "e0eaddff843a8695575daec34506f126.py", line 11, in

tuple1[1] = 100

TypeError: 'tuple' object does not support item assignment

## Accessing Values in Python Tuples

Tuples in Python provide two ways by which we can access the elements of a tuple.

* Using a positive index
* Using a negative index

### Python Access Tuple using a Positive Index

Using square brackets we can get the values from tuples in Python.

|  |
| --- |
| var **=** ("Geeks", "for", "Geeks")    print("Value in Var[0] = ", var[0])  **print**("Value in Var[1] = ", var[1])  print("Value in Var[2] = ", var[2]) |

**Output:**

Value in Var[0] = Geeks

Value in Var[1] = for

Value in Var[2] = Geeks

### Access Tuple using Negative Index

In the above methods, we use the positive index to access the value in Python, and here we will use the negative index within [].

|  |
| --- |
| var **=** (1, 2, 3)    **print**("Value in Var[-1] = ", var[**-**1])  print("Value in Var[-2] = ", var[**-**2])  print("Value in Var[-3] = ", var[**-**3]) |

**Output:**

Value in Var[-1] = 3

Value in Var[-2] = 2

Value in Var[-3] = 1

## Different Operations Related to Tuples

Below are the different operations related to tuples in Python:

* Concatenation
* Nesting
* Repetition
* Slicing
* Deleting
* Finding the length
* Multiple Data Types with tuples
* Conversion of lists to tuples
* Tuples in a Loop

### ****Concatenation of Python Tuples****

To Concatenation of Python Tuples, we will use plus operators(+).

|  |
| --- |
| # Code for concatenating 2 tuples  tuple1 **=** (0, 1, 2, 3)  tuple2 **=** ('python', 'geek')    # Concatenating above two  print(tuple1 **+** tuple2) |

**Output:**

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

### ****Nesting of Python Tuples****

A nested tuple in Python means a tuple inside another tuple.

|  |
| --- |
| # Code for creating nested tuples  tuple1 **=** (0, 1, 2, 3)  tuple2 **=** ('python', 'geek')    tuple3 **=** (tuple1, tuple2)  print(tuple3) |

**Output :**

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

### ****Repetition Python Tuples****

We can create a tuple of multiple same elements from a single element in that tuple.

|  |
| --- |
| # Code to create a tuple with repetition  tuple3 **=** ('python',)**\***3  print(tuple3) |

**Output:**

('python', 'python', 'python')

Try the above without a comma and check. You will get tuple3 as a string ‘pythonpythonpython’.

### ****Slicing Tuples in Python****

Slicing a Python tuple means dividing a tuple into small tuples using the indexing method.

|  |
| --- |
| # code to test slicing  tuple1 **=** (0 ,1, 2, 3)  print(tuple1[1:])  **print**(tuple1[::**-**1])  print(tuple1[2:4]) |

**Output:**

In this example, we sliced the tuple from index 1 to the last element. In the second print statement, we printed the tuple using reverse indexing. And in the third print statement, we printed the elements from index 2 to 4.

(1, 2, 3)

(3, 2, 1, 0)

(2, 3)

**Note:**In Python slicing, the end index provided is not included.

### ****Deleting a Tuple in Python****

In this example, we are deleting a tuple using ‘del’ keyword. The output will be in the form of error because after deleting the tuple, it will give a NameError.

**Note:**Remove individual tuple elements is not possible, but we can delete the whole Tuple using Del keyword.

|  |
| --- |
| # Code for deleting a tuple  tuple3 **=** ( 0, 1)    **del** tuple3  print(tuple3) |

**Output:**

Traceback (most recent call last):

File "d92694727db1dc9118a5250bf04dafbd.py", line 6, in <module>

print(tuple3)

**NameError**: name 'tuple3' is not defined

### ****Finding the Length of a Python Tuple****

To find the length of a tuple, we can use Python’s len() function and pass the tuple as the parameter.

|  |
| --- |
| # Code for printing the length of a tuple  tuple2 **=** ('python', 'geek')  print(len(tuple2)) |

**Output:**

2

### Multiple Data Types With Tuple

Tuples in Python are heterogeneous in nature. This means tuples support elements with multiple datatypes.

|  |
| --- |
| # tuple with different datatypes  tuple\_obj **=** ("immutable",True,23)  print(tuple\_obj) |

**Output :**

('immutable', True, 23)

### ****Converting a List to a Tuple****

We can convert a list in Python to a tuple by using the **tuple() constructor** and passing the list as its parameters.

|  |
| --- |
| # Code for converting a list and a string into a tuple  list1 **=** [0, 1, 2]    **print**(tuple(list1))    # string 'python'  print(tuple('python')) |

**Output:**

Tuples take a single parameter which may be a list, string, set, or even a dictionary(only keys are taken as elements), and converts them to a tuple.

(0, 1, 2)

('p', 'y', 't', 'h', 'o', 'n')

### ****Tuples in a Loop****

We can also create a tuple with a single element in it using loops.

|  |
| --- |
| # python code for creating tuples in a loop  tup **=** ('geek',)    # Number of time loop runs  n **=** 5  **for** i **in** range(int(n)):      tup **=** (tup,)      print(tup) |

**Output:**

(('geek',),)

((('geek',),),)

(((('geek',),),),)

((((('geek',),),),),)

(((((('geek',),),),),),)

# (3) Dictionaries in Python -

**A dictionary in Python** is a data structure that stores the value in **key:value pairs**. This makes it different from lists, tuples, and arrays as in a dictionary each key has an associated value.

***Note :*** dictionaries are ordered and can not contain duplicate keys*.*

|  |
| --- |
| Dict **=** {1: 'Geeks', 2: 'For', 3: 'Geeks'}  print(Dict) |

**Output:**

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

**dict\_var = {key1 : value1, key2 : value2, …..}**

### 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’.

**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.

|  |
| --- |
| 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 Example

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

### Different Ways to Create a Python Dictionary

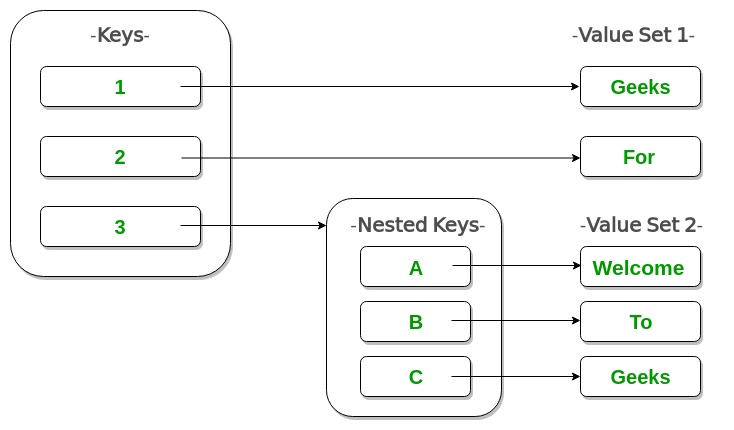
The code demonstrates different ways to create dictionaries in Python. It first creates an empty dictionary, and then shows how to create dictionaries using the **dict()**constructor with key-value pairs specified within curly braces and as a list of tuples.

|  |
| --- |
| Dict **=** {}  **print**("Empty Dictionary: ")  print(Dict)    Dict **=** dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})  **print**("\nDictionary with the use of dict(): ")  print(Dict)    Dict **=** dict([(1, 'Geeks'), (2, 'For')])  print("\nDictionary with each item as a pair: ")  print(Dict) |

**Output:**

Empty Dictionary:   
 {}  
Dictionary with the use of dict():   
{1: 'Geeks', 2: 'For', 3: 'Geeks'}  
Dictionary with each item as a pair:   
{1: 'Geeks', 2: 'For'}

## Nested Dictionaries



**Example**: The code defines a nested dictionary named **‘Dict’** with multiple levels of key-value pairs. It includes a top-level dictionary with keys 1, 2, and 3. The value associated with key 3 is another dictionary with keys ‘A,’ ‘B,’ and ‘C.’ This showcases how Python dictionaries can be nested to create hierarchical data structures.

|  |
| --- |
| Dict **=** {1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}    print(Dict) |

**Output:**

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

## Adding Elements to a Dictionary

The addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = ‘Value’.

Updating an existing value in a Dictionary can be done by using the built-in **update()** method. Nested key values can also be added to an existing Dictionary.

**Note-** While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

**Example: Add Items to a Python Dictionary with Different DataTypes**

The code starts with an empty dictionary and then adds key-value pairs to it. It demonstrates adding elements with various data types, updating a key’s value, and even nesting dictionaries within the main dictionary. The code shows how to manipulate dictionaries in Python.

|  |
| --- |
| Dict **=** {}  print("Empty Dictionary: ")  print(Dict)  Dict[0] **=** 'Geeks'  Dict[2] **=** 'For'  Dict[3] **=** 1  print("\nDictionary after adding 3 elements: ")  print(Dict)    Dict['Value\_set'] **=** 2, 3, 4  print("\nDictionary after adding 3 elements: ")  print(Dict)    Dict[2] **=** 'Welcome'  **print**("\nUpdated key value: ")  **print**(Dict)  Dict[5] **=** {'Nested': {'1': 'Life', '2': 'Geeks'}}  print("\nAdding a Nested Key: ")  print(Dict) |

**Output:**

Empty Dictionary:   
{}  
Dictionary after adding 3 elements:   
{0: 'Geeks', 2: 'For', 3: 1}  
Dictionary after adding 3 elements:   
{0: 'Geeks', 2: 'For', 3: 1, 'Value\_set': (2, 3, 4)}  
Updated key value:   
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value\_set': (2, 3, 4)}  
Adding a Nested Key:   
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value\_set': (2, 3, 4), 5:   
{'Nested': {'1': 'Life', '2': 'Geeks'}}}

## Accessing Elements of a Dictionary

To access the items of a dictionary refer to its key name. Key can be used inside square brackets.

### ****Access a Value in Python Dictionary****

The code demonstrates how to access elements in a dictionary using keys. It accesses and prints the values associated with the keys ‘name’ and 1, showcasing that keys can be of different data types (string and integer).

|  |
| --- |
| Dict **=** {1: 'Geeks', 'name': 'For', 3: 'Geeks'}  **print**("Accessing a element using key:")  print(Dict['name'])  print("Accessing a element using key:")  **print**(Dict[1]) |

**Output:**

Accessing a element using key:  
For  
Accessing a element using key:  
Geeks

There is also a method called [**get()**](https://www.geeksforgeeks.org/get-method-dictionaries-python/) that will also help in accessing the element from a dictionary. This method accepts key as argument and returns the value.

**Example: Access a Value in Dictionary using get() in Python**

The code demonstrates accessing a dictionary element using the **get()** method. It retrieves and prints the value associated with the key 3 in the dictionary **‘Dict’**. This method provides a safe way to access dictionary values, avoiding KeyError if the key doesn’t exist.

|  |
| --- |
| Dict **=** {1: 'Geeks', 'name': 'For', 3: 'Geeks'}    **print**("Accessing a element using get:")  print(Dict.get(3)) |

**Output:**

Accessing a element using get:  
Geeks

## Accessing an Element of a Nested Dictionary

To access the value of any key in the nested dictionary, use indexing [] syntax.

**Example**: The code works with nested dictionaries. It first accesses and prints the entire nested dictionary associated with the key **‘Dict1’**. Then, it accesses and prints a specific value by navigating through the nested dictionaries. Finally, it retrieves and prints the value associated with the key**‘Name’** within the nested dictionary under **‘Dict2’**.

|  |
| --- |
| Dict **=** {'Dict1': {1: 'Geeks'},          'Dict2': {'Name': 'For'}}    **print**(Dict['Dict1'])  **print**(Dict['Dict1'][1])  print(Dict['Dict2']['Name']) |

**Output:**

{1: 'Geeks'}  
Geeks  
For

## ****Deleting Elements using ‘del’ Keyword****

The items of the dictionary can be deleted by using the del keyword as given below.

**Example**: The code defines a dictionary, prints its original content, and then uses the **‘del’** statement to delete the element associated with key 1. After deletion, it prints the updated dictionary, showing that the specified element has been removed.

|  |
| --- |
| Dict **=** {1: 'Geeks', 'name': 'For', 3: 'Geeks'}    **print**("Dictionary =")  print(Dict)  **del**(Dict[1])  **print**("Data after deletion Dictionary=")  **print**(Dict) |

**Output**

Dictionary ={1: 'Geeks', 'name': 'For', 3: 'Geeks'}  
Data after deletion Dictionary={'name': 'For', 3: 'Geeks'}

## ****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. |

### 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.

|  |
| --- |
| 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 in Python

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 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**

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) |

**Output:**

Output List using list comprehensions: [2, 4, 4, 6]

**Example 1:** **Squaring Number WITHOUT using List comprehensions**

Suppose we want to create an output list which contains squares of all the numbers from 1 to 9. Let’s see how to do this using for loops and list comprehension.

## Python3

|  |
| --- |
| output\_list **=** []  **for** var **in** range(1, 10):      output\_list.append(var **\*\*** 2)    print("Output List using for loop:", output\_list) |

**Output:**

Output List using for loop: [1, 4, 9, 16, 25, 36, 49, 64, 81]

**Example 2:** **Squaring Number using List comprehensions**

In This we use list comprehension to generate a new list. It iterates through the numbers in the range from 1 to 9 (inclusive). For each number var, it calculates the square of the number using the expression and adds the result to the new list. The printed output will contain the squares of numbers from 1 to 9.

## Python3

|  |
| --- |
| list\_using\_comp **=** [var**\*\***2 **for** var **in** range(1, 10)]    **print**("Output List using list comprehension:",                                list\_using\_comp) |

**Output:**

Output List using list comprehension: [1, 4, 9, 16, 25, 36, 49, 64, 81]

## 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}

**Example 1:** **Mapping states with their capitals without Using dictionary comprehension**

Given two lists containing the names of states and their corresponding capitals, construct a dictionary which maps the states with their respective capitals. Let’s see how to do this using for loops and dictionary comprehension.

## Python3

|  |
| --- |
| state **=** ['Gujarat', 'Maharashtra', 'Rajasthan']  capital **=** ['Gandhinagar', 'Mumbai', 'Jaipur']    output\_dict **=** {}    **for** (key, value) **in** zip(state, capital):      output\_dict[key] **=** value    print("Output Dictionary using for loop:",output\_dict) |

**Output:**

Output Dictionary using for loop: {'Gujarat': 'Gandhinagar',  
 'Maharashtra': 'Mumbai',   
 'Rajasthan': 'Jaipur'}

**Example 2: Mapping states with their capitals with using dictionary comprehension**

Here we will use dictionary comprehension to initializes two lists, state and capital, containing corresponding pairs of states and their capitals. It iterates through the pairs of state and capital using the zip() function, and for each pair, it creates a key-value pair in the dictionary. The key is taken from the state list, and the value is taken from the capital list. Finally, the printed output will contain the mapping of states to their capitals.

## Python3

|  |
| --- |
| state **=** ['Gujarat', 'Maharashtra', 'Rajasthan']  capital **=** ['Gandhinagar', 'Mumbai', 'Jaipur']    dict\_using\_comp **=** {key:value **for** (key, value) **in** zip(state, capital)}    **print**("Output Dictionary using dictionary comprehensions:",                                             dict\_using\_comp) |

**Output:**

Output Dictionary using dictionary comprehensions: {'Rajasthan': 'Jaipur',  
 'Maharashtra': 'Mumbai',  
 'Gujarat': 'Gandhinagar'}

## 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}

**Generator Comprehensions**

Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets. The major difference between them is that generators don’t allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient. Let’s look at the following example to understand generator comprehension:

**Python3**

|  |
| --- |
| input\_list **=** [1, 2, 3, 4, 4, 5, 6, 7, 7]    output\_gen **=** (var **for** var **in** input\_list **if** var **%** 2 **==** 0)    **print**("Output values using generator comprehensions:", end **=** ' ')    **for** var **in** output\_gen:      print(var, end **=** ' ') |

**Output:**

Output values using generator comprehensions: 2 4 4 6

**Comparison Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Basis** | **List** | **Tuple** | **Set** | **Dictionary** |
| Syntax | [ ] | ( ) | { } | { } |
| Mutable/Immutable | Mutable | Immutable | Mutable | Mutable |
| Order | Ordered | Ordered | Unordered | Unordered |
| Duplicates | Allowed | Allowed | Not Allowed | Not Allowed |
| Indexing | Allowed | Allowed | Not Allowed | Allowed |
| Slicing | Allowed | Allowed | Not Allowed | Not Allowed |
| Common Operations | append(), insert(), remove(), pop(), extend() | Concatenation, unpacking, indexing, slicing | add(), remove(), union(), intersection(), difference() | keys(), values(), items(), get() |
| Applications | Storing mutable sequences of items | Storing immutable sequences of items, returning multiple values from a function | Performing set operations, removing duplicates from a list | Storing key-value pairs, providing structured access to data |
| Limitations | Slow when dealing with large lists, takes up more memory compared to tuples | Cannot add, remove or modify elements after creation | Does not preserve order, cannot store duplicates | Keys must be unique and immutable, values can be mutable or immutable |

**Difference between List, Tuple, Set, and Dictionary**

| **List** | **Tuple** | **Set** | **Dictionary** |
| --- | --- | --- | --- |
| A list is a **non-homogeneous** data structure that **stores the elements in columns of a single row or multiple rows**. | A Tuple is also a **non-homogeneous** data structure that **stores elements in columns of a single row or multiple rows.** | The set data structure is also a **non-homogeneous** data structure but **stores the elements in a single row**. | A dictionary is also a **non-homogeneous** data structure that **stores key-value pairs**. |
| The list can be represented by [ ] | Tuple can be represented by  ( ) | The set can be represented by { } | The dictionary can be represented by { } |
| The list **allows duplicate elements** | Tuple **allows duplicate elements** | The Set **will not allow duplicate elements** | The dictionary **doesn’t allow duplicate keys**. |
| The list can use nested among all | Tuple can use nested among all | The set can use nested among all | The dictionary can use nested among all |
| Example: [1, 2, 3, 4, 5] | Example: (1, 2, 3, 4, 5) | Example: {1, 2, 3, 4, 5} | Example: {1: “a”, 2: “b”, 3: “c”, 4: “d”, 5: “e”} |
| A list can be created using the **list()**function | Tuple can be created using the **tuple()** function. | A set can be created using the **set()** function | A dictionary can be created using the **dict()**function. |
| A list is **mutable** i.e we can make any changes in the list. | A tuple is **immutable** i.e we can not make any changes in the tuple. | A set is **mutable** i.e we can make any changes in the set, its elements are not duplicated. | A dictionary is **mutable**, its Keys are **not duplicated**. |
| List is **ordered** | Tuple is **ordered** | Set is **unordered** | Dictionary is **unordered** |
| Creating an empty list  l=[] | Creating an empty Tuple  t=() | Creating a set  a=set() b=set(a) | Creating an empty dictionary  d={} |

## ****Differences between List and Tuple in Python****

| **Sno** | **LIST** | **TUPLE** |
| --- | --- | --- |
| 1 | Lists are **mutable** | Tuples are **immutable** |
| 2 | The implication of iterations is **Time-consuming** | The implication of iterations is **comparatively Faster** |
| 3 | The list is better for performing operations, such as **insertion and deletion**. | A Tuple data type is appropriate for **accessing the elements** |
| 4 | Lists **consume more memory** | Tuple **consumes less memory** as compared to the list |
| 5 | Lists have **several built-in methods** | Tuple **does not have many built-in methods**. |
| 6 | **Unexpected changes and errors** are more likely to occur | In a tuple, it is **hard to take place.** |