# In the earlier classes of **Software testing**, we learned that software testing is classified into two types of testing, which are **Manual testing and Automation testing**. Both manual and automation testing have their characteristics and approaches that make both the testing technique different from each other.

# What Is Automation Testing?

Automation testing refers to the automatic testing of the software in which tester write the test script with the help of testing tools and framework and run it on the software. The test script automatically test the software without human intervention and shows the result (either error, bugs are present or software is free from them).

Automation testing needs manual effort when creating initial scripts, and further process is performed automatically to compare the actual testing result with expected results.

## **Why do Companies perform Automation Testing?**

In software testing, automation testing is required to test the application because it offers us a better application with less effort and time.

* Reusability
* Consistency
* Running tests anytime (24/7)
* Early Bug detection
* Less Human Resources

|  |  |
| --- | --- |
| **Manual testing** | **Automation testing** |
| Testing in which a human tester executes test cases | In automation testing, automation tools are used to execute the test cases |
| In this testing, human resources are involved, that's why it is time-consuming | It is much faster than the manual testing |
| It is repetitive and error-prone | Here automated tools are used that make it interesting and accurate |
| BVT (build verification testing) is time-consuming and tough in manual testing | It's easy to build verification testing |
| Instead of frameworks, this testing use checklist, guidelines, and stringent process for drafting test cases. | Frameworks like keyword, hybrid, and data drive to accelerate the automation process. |
| The process turnaround time is higher than the automation testing process (one testing cycle takes lots of time) | It completes a single round of testing within record time; therefore, a process turnaround time is much lower than a manual testing process. |
| It is best for usability, exploratory and adhoc testing | It is widely used for GUI testing, Regression testing and Performance testing. |
| Low return on investment | The high return on investment |

There are many languages for humans to communicate with each other. Like English Hindi Marathi Tamil Telugu etc… These are used for proper communication among humans they should understand what kind of questions are asking the person and what I need to respond to that question.

Same Applies for Computer Programming Language.

There will set of instructions will be written by programmers which computer needs to understand and it should respond to that instruction.

# What Is Computer Programming Language?

Computer programming languages allow user to give instructions to a computer in a language the computer understands.

Computer programming is the act of writing computer programs, which are a sequence of instructions written using a Computer Programming Language to perform a specified task by the computer.

There are different languages:

1. Machine Language (Low Level Language)

All the instructions of machine language are written in the form of binary numbers 1's & 0's

This is the language that is written for the computer hardware. Such language is effected directly by the central processing unit (CPU) of a computer system.

1. Assembly Language (Middle Level Language)

Middle-level language is a computer language in which the instructions are created using symbols such as letters, digits and special characters. **Assembly language** is an example of middle-level language. In assembly language, we use predefined words called mnemonics. Binary code instructions in low-level language are replaced with mnemonics and operands in middle-level language. But the computer cannot understand mnemonics, so we use a translator called **Assembler** to translate mnemonics into machine language.

1. High Level Language

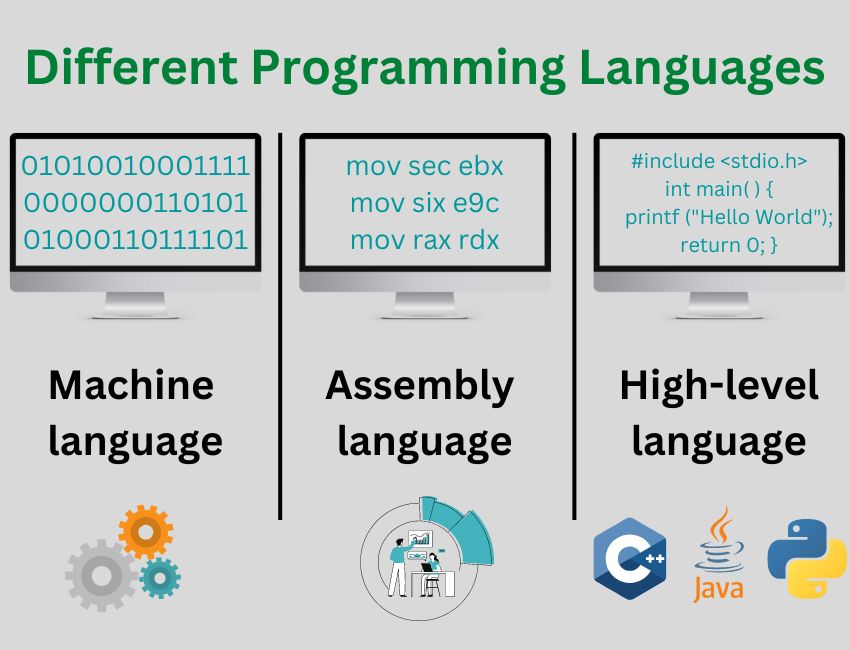
High-level language is a computer language which can be understood by the users. The high-level language is very similar to human languages and has a set of grammar rules that are used to make instructions more easily. Every high-level language has a set of predefined words known as Keywords and a set of rules known as Syntax to create instructions. The high-level language is easier to understand for the users but the computer can not understand it. High-level language needs to be converted into the low-level language to make it understandable by the computer HARDWARE.Top of Form

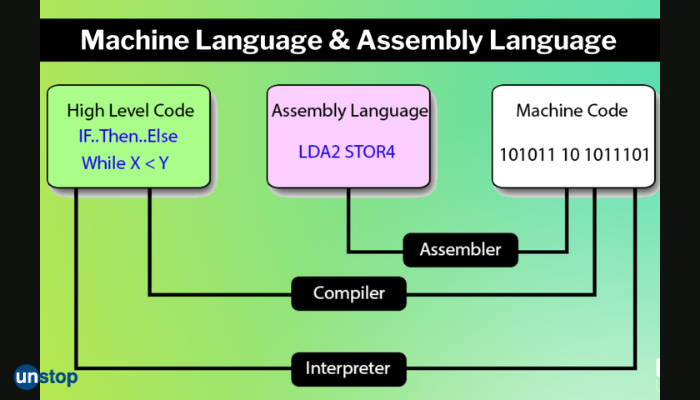
Languages like FORTRAN,C, C++, JAVA, Python, DOT NET etc., are examples of high-level languages. All these programming languages use human-understandable language like English to write program instructions. These instructions are converted to low-level language by the compiler or interperter so that it can be understood by the computer.

Most of the Human Interface Languages (Hindi, English, Spanish, French, etc.) are made of several elements like verbs, nouns, adjectives, adverbs, propositions, and conjunctions, etc.

Similar to Human Interface Languages, Computer Programming Languages are also made of several elements. These basic elements include −

* Programming Environment
* Basic Syntax
* Data Types
* Variables
* Keywords
* Basic Operators
* Decision Making
* Loops
* Numbers
* Characters
* Strings
* Functions
* File I/O





**Compiler And Interpreter:**

We generally write a computer program using a high-level language. A high-level language is one that is understandable by us, humans. This is called **source code**.

However, a computer does not understand high-level language. It only understands the program written in **0**'s and **1**'s in binary, called the **machine code**.

To convert source code into machine code, we use either a **compiler** or an **interpreter**.

Both compilers and interpreters are used to convert a program written in a high-level language into machine code understood by computers. However, there are differences between how an interpreter and a compiler works.

|  |  |
| --- | --- |
| Interpreter | Compiler |
| Translates program one statement at a time. | Scans the entire program and translates it as a whole into machine code. |
| Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers. | Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters. |
| No Object Code is generated, hence are memory efficient. | Generates Object Code which further requires linking, hence requires more memory. |
| Programming languages like JavaScript, Python, Ruby use interpreters. | Programming languages like C, C++, Java use compilers. |

**Introduction of Python**

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages

Python is a cross-platform programming language, which means that it can run on multiple platforms like Windows, macOS, Linux, and has even been ported to the Java and .NET virtual machines. It is free and open-source

**Python is a general purpose high level programming language(applicable for wide range applications)**

**Python History**

**Python was developed by Guido Van Rossam in 1989 while working at National**

**Research Institute at Netherlands.**

**Java 1995--- python is older than Java.**

**But officially Python was made available to public in 1991. The official Date of Birth for**

**Python is : Feb 20th 1991.**

**The name Python was selected from the TV Show**

**"The Complete**

**Monty**

**Python's**

**Circus", which was broadcasted in BBC from 1969 to 1974.**

**Guido developed Python language by taking almost all programming features from**

**different languages**

**1. Functional Programming Features from C**

**2. Object Oriented Programming Features from C++**

**3. Scripting Language Features from Perl and Shell Script**

**4. Modular Programming Features from Modula-3**

**Most of syntax in Python Derived from C and ABC languages.**

**Python Features**

1. **Simple and easy to learn:**

Python is a simple programming language. When we read Python program,we can feel like

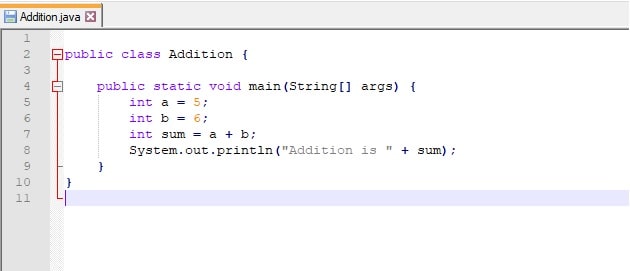
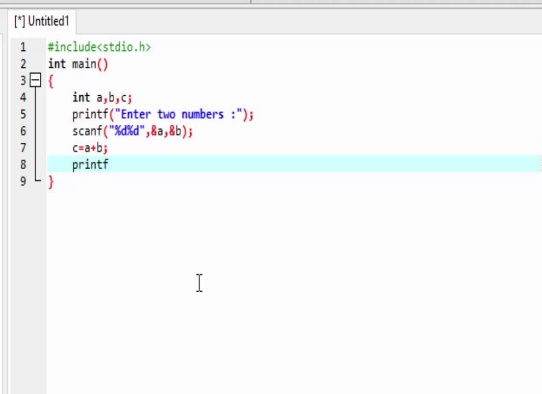
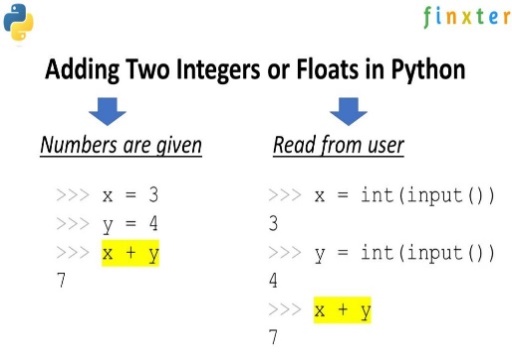
reading english statements.

The syntaxes are very simple and only 30+ kerywords are available.

When compared with other languages, we can write programs with very less number of

lines. Hence more readability and simplicity.

We can reduce development and cost of the project.

1. **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
2. **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code
3. **Platform Independent:** Once we write a Python program,it can run on any platform without rewriting once again.

**5. Portability:**

Python programs are portable. ie we can migrate from one platform to another platform

very easily. Python programs will provide same results on any paltform.

1. **Dynamically Typed:**

In Python we are not required to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically. Hence Python is considered as dynamically typed language. But Java, C etc are Statically Typed Languages b'z we have to provide data type at the beginning only. This dynamic typing nature will provide more flexibility to the programmer

1. **Both Procedure Oriented and Object Oriented:**

Python language supports both Procedure oriented (like C, pascal etc) and object oriented (like C++,Java) features. Hence we can get benefits of both like security and reusability etc.

8. **Interpreted**:

We are not required to compile Python programs explcitly. Internally Python interpreter will take care that compilation.

If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

**Where we can use Python:**

We can use everywhere. The most common important application areas are

1. For developing Desktop Applications

2. For developing Web Applications

3. For developing database Applications

4. For Network Programming

5. For developing games

6. For Data Analysis /Data Science Applications

**7. For Machine Learning**

**8. For developing Artificial Intelligence Applications**

9. For IOT

**Different ways to write a python programs and run:**

1. **IDLE**

IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python. The Python installer for Windows contains the IDLE module by default.

IDLE is not available by default in Python distributions for Linux. It needs to be installed using the respective package managers. Execute the following command to install IDLE on Ubuntu:

$ sudo apt-get install idle

1. **CMD**
2. **Notepad**
3. **Pycharm IDE (Integrated Development Environment)**

print('Yusuf Tamboli');  
a=10  
b=20  
print(a+b);  
print(100+10);

**Run this code on above all IDE’**

**Python Syntax**

The Python syntax defines all the set of rules that are used to create sentences in Python programming.

**For example –** We have to learn grammar to learn the English language. In the same way, you will need to learn and understand the Python syntax in order to learn the Python language.

1. **Line Structure**
2. Print data in Single line

Syntax- print(“Hello All”)

1. Print data in next lines

Syntax-

print("Hello Guys\n"   
 "How r u")

print("Hello Guys\n" "How r u")

print("""Hi  
how are you?"""

print("Hello Guys");print("How r u") –not efficient way

1. If you run below statement, it will be displayed in same line.

print("Hello Guys"

"How r u")

1. **Python Comments**

A comment is a programmer-readable explaination 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.

Single line Comment

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

#print(“Hello All”)—It will not print

print("Hello All") #“how r u” It will print Hello All and ignore how r u.

Multiline Comment

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

'''  
print("Hello All")  
print("How r u")  
print("All Good?")  
'''  
print("All Good?")

It’will print only All Good? And ignore all other statemnts.

You can use double quote too

"""  
print("Hello All")  
print("How r u")  
print("All Good?")  
"""  
print("All Good?"

**C. Python Quotations**

Python supports the single quote and the double quote for string literals. But if you begin a string with a single quote, you must end it with a single quote only.

The same goes for double-quotes.

‘Yusuf’ “Yusuf”

The following string is delimited by single quotes.

>>> print(‘We need a chaperone');

**Output:**

We need a chaperone

Below string is delimited by double-quotes.

>>> print("We need a 'chaperone'");

**Output:**

We need a ‘chaperone’

## **D. Python Indentation**

Since Python doesn’t use curly braces to delimit blocks of code, this Python Syntax is mandatory.

You can indent code under a function, loop, or class.

'''  
if 2>1:  
print("2 is the bigger person");  
  
'''--- It will not work becz of indents are not given

if 2>1:  
 print("2 is the bigger 1");--- It will work

## **E. Python Keywords**

## Keywords are predefined, reserved words used in Python programming that have special meanings to the interpreter.

We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language.

All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

**import keyword  
print(keyword.kwlist)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **Python Keywords List** |  |  |
| False | Await | Else | import | pass |
| None | Break | Except | in | raise |
| True | Class | Finally | is | return |
| and | Continue | For | lambda | try |
| as | Def | From | nonlocal | while |
| assert | Del | Global | not | with |
| async | elif | If | or | yield |

## **F. Python Identifiers**

**Identifier**is a user-defined name given to a variable, function, class, module, etc. The identifier is a combination of character digits and an underscore. They are case-sensitive i.e., ‘Name and ‘name’, and ‘NAME’ are three different identifiers in python.

All the variables, class, object, functions, lists, dictionaries etc. in Python are together termed as Python Identifiers. Identifiers are the basis of any Python program. Almost every Python Code uses some or other identifiers.

That means whenever we want to give an entity a name, that’s called identifier.

## Rules for Naming Python Identifiers

* It cannot be a reserved python keyword.
* It should not contain white space.
* It can be a combination of A-Z, a-z, 0-9, or underscore.
* It should start with an alphabet character or an underscore ( \_ ).
* It should not contain any special character other than an underscore ( \_ )
* We cannot use special symbols like **!**, **@**, **#**, **$**, and so on
* The first letter of an identifier cannot be a digit.

Though Above are hard rules for writing identifiers, also there are some naming conventions which are not mandatory but rather good practices to follow.

1. Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
2. Starting an identifier with a single leading underscore indicates the identifier is private.
3. If the identifier starts and ends with two underscores, that means the identifier is language-defined special name.
4. While **c = 10** is valid, writing **count = 10** would make more sense and it would be easier to figure out what it does even when you look at your code after a long time.
5. Multiple words can be separated using an underscore, for example **this\_is\_a\_variable**

Examples

1.

myVariable="hello world"

print(myVariable)

var1=1

print(var1)

var2=2

print(var2)

2.

name = ‘Yusuf’

Here, name is a variable (an identifier) which holds the value 'Yusuf'.

3. We cant use reserved keywords as identifier in python.

For example,

continue = ‘Yusuf’

if= ‘abc’

import,try etc…

The above code is wrong because we have used continue as a variable name.

### Some Valid and Invalid Identifiers in Python

|  |  |
| --- | --- |
| Valid Identifiers | Invalid Identifiers |
| score | @core |
| return\_value | return |
| highest\_score | highest score |
| name1 | 1name |
| convert\_to\_string | convert to\_string |

## **G .** **Python Variables**

Python variables are the reserved memory locations used to store values within a Python Program. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.

Python variable is also known as an identifier and used to hold a value.

In programming, a variable is a container (storage area) to hold data. For example,

number = 10

Here, number is the variable storing the value **10**.

## Assigning values to Variables in Python

As we can see from the above example, we use the assignment operator = to assign a value to a variable.

# assign value to site\_name variable

site\_name = 'cred.pro'

print(site\_name)

# Output: cred.pro

In the above example, we assigned the value cred.com to the site\_name variable. Then, we printed out the value assigned to site\_name.

## Changing the Value of a Variable in Python

site\_name = 'cred.com'

print(site\_name)

# assigning a new value to site\_name

site\_name = 'apple.com'

print(site\_name)

**Output**

cred.com

apple.com

Here, the value of site\_name is changed from 'cred.com to 'apple.com'.

### Example: Assigning multiple values to multiple variables

a, b, c = 5, 3.2, 'Hello'

print(a) # prints 5

print(b) # prints 3.2

print(c) # prints Hello

If we want to assign the same value to multiple variables at once, we can do this as:

name1 = name2 = ‘cred.com'

print(site1) # prints cred.com

print(site2) # prints cred.com

Here, we have assigned the same string value cred.com. to both the

variables name1 and name2 .

### Swapping Variables

Swapping means interchanging values. To swap Python variables, you don’t need to do much.

>>> a,b='red','blue'

>>> a,b=b,a

>>> print(a,b)

**Output**

blue red

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X | y |  |  | y | x |
| Yusuf | Tushar |  |  | yusuf | Tushar |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| X | y |  |  | x | y |
| Yusuf | Tushar |  |  | Tushar | yusuf |

### Deleting Variables

You can also delete Python variables using the keyword ‘del’.

>>> a='red'

>>> del a

>>> print(a)

**Python Variable Types**

**Global**- It will define globally and anywhere we can use it.

a=10 , Name=’Yusuf’

**Local**- It will define in function and we can access then in function only

def add():  
 a=10  
 b=20  
 print(a+b)

**The rules to name Variables are given below.**

* The first character of the variable must be an alphabet or underscore ( \_ ).
* All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
* Variables name must not contain any white-space, or special character (!, @, #, %, ^, &, \*).
* Variables name must not be similar to any keyword defined in the language.
* Variables names are case sensitive; for example, myname, and MyName is not the same.
* Examples of valid identifiers: a123, \_n, n\_9, etc.
* Examples of invalid identifiers: 1a, n%4, n 9, etc

## **I.** **Python Data Types**

Although we don’t have to declare a type for Python variables, a value does have a type. This information is vital to the interpreter.

Python supports the following data types.

### 1. Python Numbers

There are four numeric Python data types.

#### a. int

int stands for integer. This Python Data Type holds signed integers. We can use the type() function to find which class it belongs to.

>>> a=-7

>>> print(type(a))

**Output**

**<class ‘int’>**

An integer can be of any length, with the only limitation being the available memory.

>>> a=9999999999999999999999999999999

>>> print(type(a))

**Output**

**<class ‘int’>**

#### b. float

This Python Data Type holds floating-point real values. An int can only store the number 3, but float can store 3.25 if you want.

>>> a=3.0

>>> print(type(a))

**Output**

**<class ‘float’>**

#### c. long

This Python Data type holds a long integer of unlimited length. But this construct does not exist in Python 3.x.

#### d. complex

This Python Data type holds a complex number. A complex number looks like this: a+bj Here, a and b are the real parts of the number, and j is imaginary.

>>> a=2+3j

>>> print(type(a))

**Output**

**<class ‘complex’>**

### 2. Strings

A string is a sequence of characters. Python does not have a char data type, unlike C++ or Java. You can delimit a string using single quotes or double-quotes.

>>> city='Ahmedabad'

>>> print(city)

**Output**

**‘Ahmedabad’**

>>> city="Ahmedabad"

>>> print(type(city))

Output-

<class 'str'>

How to check data type and address of variable?

**id() Function:**

id() is an inbuilt function in Python.

### Syntax :

id(object)

Returns the identity of object. It is the address of object in memory.

It will be unique and constant throughout the lifetime of object.

a=10  
print(id (a))

# a=10  
# print(id (a))  
# print(type (a))  
# a,b,c= 10,20,30  
# print(id (a))

# print(id (b))  
# print(id (c))  
  
# a=10  
# print(id (a))  
# print(type (a))  
# a,b,c= 10,10,20  
# print(id (a))  
# print(id (b))  
# print(id (c))  
  
# a=10  
# print(id (a))  
# print(type (a))  
# a,b,c= 10,10,10  
# print(id (a))  
# print(id (b))  
# print(id (c))



|  |  |  |  |
| --- | --- | --- | --- |
| A | 10 |  | 140723236562312 |
| A | 20 |  | 140723236561992 |
| C | 10 |  | 140723236562312 |
| D | 20 |  | 140723236561992 |
| E | ‘Yusuf’ |  | 1760708679216 |
| F |  |  |  |
|  |  |  |  |
|  |  |  |  |

**type() Function:**

type(object) is an inbuilt function in Python.

The type() function returns the data type of the specified object

a=100  
print(type (a))

Output <class 'int'>

**J.** **Python Literals**

# a=10  
# name='yusuf'  
# c=10.5  
# d=100+10j

10- int literal

Yusuf- string literal

10.5 – float literal

100+10j- complex literal

**Difference between identifiers Variables Data Types**

**Candiadtename** =’Yusuf’---- candidatename is ur **variable** which holds the Yusuf string. Which has string datatype.

**Candiadteid** =101—candidateid is also ur **variable** which holds the 101 number. Which has int datatype.

But both these variables are identifies by different names so we can call it them as **identifiers.**

# Type Casting in Python/ Type Conversion in Python

In programming, type conversion is the process of converting data of one type to another. For example: converting int data to str.

Type Casting is the method to convert the variable data type into a certain data type in order to the operation required to be performed by users.

There are two types of type conversion in Python.

* Implicit Conversion/Type Casting - automatic type conversion
* Explicit Conversion/Type Casting - manual type conversion

## **Implicit Type Conversion**

In this,  methods, Python converts data type into another data type automatically. In this process, users don’t have to involve in this process.

# Python program to demonstrate

# implicit type Casting

# Python automatically converts a to int

a = 7

print(type(a))

# Python automatically converts b to float

b = 3.0

print(type(b))

# Python automatically converts c to float as it is a float addition

c = a + b

print(c)

print(type(c))

# Python automatically converts d to float as it is a float multiplication

d = a \* b

print(d)

print(type(d))

## Example 1: Converting integer to float

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

integer\_number = 123

float\_number = 1.23

new\_number = integer\_number + float\_number

# display new value and resulting data type

print("Value:",new\_number)

print("Data Type:",type(new\_number))

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

**Output**

Value: 124.23

Data Type: <class 'float'>

In the above example, we have created two variables: integer\_number and float\_number of int and float type respectively.

Then we added these two variables and stored the result in new\_number.

As we can see new\_number has value **124.23** and is of the float data type.

It is because Python always converts smaller data types to larger data types to avoid the loss of data.

## **Explicit Type Casting**

In this method, Python need user involvement to convert the variable data type into certain data type in order to the operation required.

Mainly in type casting can be done with these data type function:

* **Int() :**Int() function take float or string as an argument and return int type object.
* **float() :**float() function take int or string as an argument and return float type object.
* **str() :**str() function take float or int as an argument and return string type object.

### Let’s see some example of type casting:

**Type Casting int to float:**

Here, we are casting integer object to float object with **float()** function.

# Python program to demonstrate

# type Casting

# int variable

a **=** 5

# typecast to float

n **=** float(a)

print(n)

print(type(n))

**Output:**

5.0

<class 'float'>

**Type Casting float to int:**

Here, we are casting float data type into integer data type with **int()** function.

# Python program to demonstrate

# type Casting

# int variable

a **=** 5.9

# typecast to int

n **=** int(a)

print(n)

print(type(n))

**Output:**

5

<class 'int'>

**Type casting int to string:**

Here, we are casting int data type into string data type with **str()** function

# Python program to demonstrate

# type Casting

# int variable

a **=** 5

# typecast to str

n **=** str(a)

print(n)

print(type(n))

**Output:**

5

<class 'str'>

**Type Casting string to int:**

Here, we are casting string data type into integer data type with **int()** function.

# Python program to demonstrate

# type Casting

# string variable

a **=** "5"

# typecast to int

n **=** int(a)

print(n)

print(type(n))

**Output:**

5

<class 'int'>

**Type Casting String to float:**

Here, we are casting string data type into float data type with **float()** function.

# Python program to demonstrate

type Casting

# string variable

a **=** "5.9"

# typecast to float

n **=** float(a)

print(n)

print(type(n))

**Output:**

5.9

<class 'float'>

**Python Operators**

In Python programming, Operators in general are used to perform operations on values and variables. These are symbols used for the purpose of logical, arithmetic and various other operations.

* OPERATORS: These are the special symbols. Eg- + , \* , /, etc.
* OPERAND: It is the value on which the operator is applied.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called **operands** and + is called **operator**.

## **Types of Python Operators**

Python language supports the following types of operators.

* Arithmetic Operators
* Comparison (Relational) Operators
* Logical Operators
* Assignment Operators
* Membership Operators
* Identity Operators

## **Arithmetic Operators in Python**

Python Arithmetic operators are used to perform basic mathematical operations like**addition, subtraction, multiplication**, and **division**.

In Python 3.x the result of division is a floating-point while in Python 2.x division of 2 integers was an integer. To obtain an integer result in Python 3.x floored (// integer) is used.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| + | Addition | 10 + 20 = 30 |
| - | Subtraction | 20 – 10 = 10 |
| \* | Multiplication | 10 \* 20 = 200 |
| / | Division | 20 / 10 = 2 |
| % | Modulus | 22 % 10 = 2 |
| \*\* | Exponent | 4\*\*3 = 16 |
| // | Floor Division | 9//2 = 4 |
| # Examples of Arithmetic Operator  a **=** 9  b **=** 4    # Addition of numbers  add **=** a **+** b    # Subtraction of numbers  sub **=** a **-** b    # Multiplication of number  mul **=** a **\*** b    # Modulo of both number  mod **=** a **%** b    # Power  p **=** a **\*\*** b    # print results  print(add)  print(sub)  **print**(mul)  print(mod)  print(p) | | |

**Output:**

13

5



36



1

6561



**Division Operators**allow you to divide two numbers and return a quotient, i.e., the first number or number at the left is divided by the second number or number at the right and returns the quotient.

There are two types of division operators:

1. Float division
2. Floor division

#### **Float division**

The quotient returned by this operator is always a float number, no matter if two numbers are integers. For example:

|  |
| --- |
| # python program to demonstrate the use of "/"  print(5**/**5)  print(10**/**2)  print(**-**10**/**2)  print(20.0**/**2) |

**Output:**

1.0

5.0

-5.0

10.0

#### **Integer division( Floor division)**

The quotient returned by this operator is dependent on the argument being passed. If any of the numbers is float, it returns output in float. It is also known as Floor division because, if any number is negative, then the output will be floored.

When the result of floor division (//) is positive, it is as though the fractional portion is truncated off, leaving only the integer portion. When the result is negative, the result is rounded down to the next smallest (greater negative) integer:

For example:

|  |
| --- |
| >>> 10 / 4  2.5  >>> 10 // 4  2  >>> 10 // -4  -3  >>> -10 // 4  -3  >>> -10 // -4  2 |
| # python program to demonstrate the use of "//"  **print**(10**//**3)  **print** (**-**5**//**2)  print (5.0**//**2)  print (**-**5.0**//**2) |

**Output:**

3

-3

2.0

-3.0

### Precedence of Arithmetic Operators in Python.The precedence of Arithmetic Operators in python is as follows:

1. P – Parentheses
2. E – Exponentiation
3. M – Multiplication (Multiplication and division have the same precedence)
4. D – Division
5. A – Addition (Addition and subtraction have the same precedence)
6. S – Subtraction

The modulus operator helps us extract the last digit/s of a number. For example:

* x % 10 -> yields the last digit
* x % 100 -> yield last two digits

## **Python Comparison Operators**

Comparison operators compare two values/variables and return a boolean result: True or False. For example,

a = 5

b =2

print (a > b) # True

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| == | Is Equal To | 3 == 5 gives us **False** |
| != | Not Equal To | 3 != 5 gives us **True** |
| > | Greater Than | 3 > 5 gives us **False** |
| < | Less Than | 3 < 5 gives us **True** |
| >= | Greater Than or Equal To | 3 >= 5 give us **False** |
| <= | Less Than or Equal To | 3 <= 5 gives us **True** |

### Example 3: Comparison Operators

# Examples of Relational Operators

a **=** 13

b **=** 33

# a > b is False

**print**(a > b)

# a < b is True

**print**(a < b)

# a == b is False

print(a **==** b)

# a != b is True

**print**(a !**=** b)

# a >= b is False

**print**(a >**=** b)

# a <= b is True

**print**(a <**=** b)

**Output**

False

True

False

True

False

True

### Precedence of Comparison Operators in Python

In python, the comparison operators have lower precedence than the arithmetic operators. All the operators within comparison operators have same precedence order.

## **Logical Operators in Python**

Python Logical operators perform **Logical AND**, **Logical OR**, and**Logical NOT** operations. It is used to combine conditional statements.

The assessment of expressions to make decisions typically makes use of the logical operators. The following logical operators are supported by Python.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| And | The condition will also be true if the expression is true. If the two expressions a and b are the same, then a and b must both be true. |
| Or | The condition will be true if one of the phrases is true. If a and b are the two expressions, then an or b must be true if and is true and b is false. |
| Not | If an expression **a** is true, then not (a) will be false and vice versa. |

 For example,

# Examples of Logical Operator

a **=** True

b **=** False

# Print a and b is False

**print**(a **and** b)

# Print a or b is True

**print**(a **or** b)

# Print not a is False

**print**(**not** a)

a = 5

b = 6

print((a > 2) and (b >= 6)) # True

print((a > 6) and (b >= 2)) # False

print((a > 2) or (b >= 6)) # True

print((a > 6) or (b >= 2)) # True

print((a > 6) and (b >= 16)) # False

print (not a) # False

x = 5  
print(not x < 10) # False

### Precedence of Logical Operators in Python

The precedence of Logical Operators in python is as follows:

1. Logical not
2. logical and
3. logical or

## **Python Assignment Operators**

Python assignment operators are used to assign values to variables. These operators include simple assignment operator, addition assign, subtraction assign, multiplication assign, division and assign operators etc.

Assignment operators are used to assign values to variables. For example,

# assign 5 to x

var x = 5

Here, = is an assignment operator that assigns 5 to x.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| = | Assignment Operator | a = 10 |
| += | Addition Assignment | a += 5 (Same as a = a + 5) |
| -= | Subtraction Assignment | a -= 5 (Same as a = a - 5) |
| \*= | Multiplication Assignment | a \*= 5 (Same as a = a \* 5) |
| /= | Division Assignment | a /= 5 (Same as a = a / 5) |
| %= | Remainder Assignment | a %= 5 (Same as a = a % 5) |
| \*\*= | Exponent Assignment | a \*\*= 2 (Same as a = a \*\* 2) |
| //= | Floor Division Assignment | a //= 3 (Same as a = a // 3) |

# Examples of Assignment operators  
a = 100  
b = 50  
# a+=15  
# print(a)  
# b+=5  
# print(b)  
# a-=15  
# print(a)  
# b-=5  
# print(b)  
# a\*=15  
# print(a)  
# b\*=5  
# print(b)  
# a/=15  
# print(a)  
# b/=5  
# print(b)  
# a%=15  
# print(a)  
# b%=5  
# print(b)  
# a//=15  
# print(a)  
# b//=5  
# print(b)

## **Identity Operators in Python**

In Python, **is** and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

**is** True if the operands are identical

**is not** True if the operands are not identical

### **Example Identity Operators in Python**

Let’s see an example of Identity Operators in Python.

a **=** 10

b **=** 20

c **=** a #(10)

print(a **is** **not** b)

print(a **is** c)

**Output**

True

True

# a=10  
# b=20  
# c=a  
# print(a is b)  
# print(a is c)  
# print(a is not b)  
# print(a is not c)

### Membership operators

In Python, in and not in are the membership operators. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| In | True if value/variable is **found** in the sequence | 5 in x |
| not in | True if value/variable is **not found** in the sequence | 5 not in x |

# x = 'Hello world'  
# print('H' in x)  
# print('h' in x)  
# print('Hello' in x)  
# print('Hello' not in x)

# list1= [1,2,3,4,5]  
# string1= "My name is AskPython"  
# tuple1=(11,22,33,44)  
#  
# print(5 in list1) #True  
# print("is" in string1) #True  
# print(88 in tuple1) #False  
#  
# print(5 not in list1) #False  
# print("is" not in string1) #False  
# print(88 not in tuple1) #True

# Python User Input-

How to take input from user?

**Python input() Function**

Python input() function is used to get input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns that.

The input() function takes input from the user and returns it.

**User Input**

Python allows for user input.

That means we are able to ask the user for input.

**How input() function works?**

* The flow of the program has stopped until the user enters the input.
* The text statement which also knows as prompt is optional to write in **input()** function. This prompt will display the message on the console.
* The **input()** function automatically converts the user input into string. We need to explicitly convert the input using the type casting.

**Syntax**

Input()

input ([prompt])

**Parameters**

**prompt**: It is a string message which prompts for the user input.

**Return**

It returns user input after converting into a string.

Examples

# name1= input()  
# print (name1)  
  
# name1= input()  
# print ("My Name is " + name1)

# name1= input()  
# name2= input()  
# print ("My Name is " + name1, name2)  
  
  
# print("a","b","c")  
# print("a"+"b"+"c")

Get input from user with a prompt

# name = input("Enter your name:")  
# print(name)

# username = input("Enter username:")  
# print("Username is " + username)

# name = input("Enter your name: ") # String Input  
# age = int(input("Enter your age: ")) # Integer Input  
# marks = float(input("Enter your marks: ")) # Float Input

# print("The name is:", name)  
# print("The age is:", age)  
# print("The marks is:", marks)

Type Casting in Input function

# val = input("Enter an integer: ")  
# # Displaying result  
# val = int(val) # casting into string  
# sqr = (val\*val) # getting square  
# print("Square of the value:",sqr)

# num1=input()  
# num2=input()  
# print(num1+num2)  
# print(type((num1+num2)))

# num1=int(input("Enter number1: "))  
# num2=int(input("Enter number2: "))  
# print("Sum of number1 and number2 is:", int(num1+num2))  
# print(type((num1+num2)))

**[Python Data Types](https://www.softwaretestinghelp.com/python-data-types/" \l "Python_Data_Types)**

Python Data Types are used to define the type of a variable. It defines what type of data we are going to store in a variable. The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various built-in data types which we will discuss with in this tutorial:

* Numeric - int, float, complex
* String/Text – str( also comes under sequence )
* Sequence - list, tuple, range
* Mapping - dict
* Boolean - bool
* Set - set, frozenset
* None – NoneType

Python Collections- list tuple set dict



**1. Numeric (Numbers)**

#### a. int

int stands for integer. This Python Data Type holds signed integers. We can use the type() function to find which class it belongs to.

>>> a=-7

>>> print(type(a))

**Output**

**<class ‘int’>**

An integer can be of any length, with the only limitation being the available memory.

>>> a=9999999999999999999999999999999

>>> print(type(a))

**Output**

**<class ‘int’>**

#### b. float

This Python Data Type holds floating-point real values. An int can only store the number 3, but float can store 3.25 if you want.

>>> a=3.0

>>> print(type(a))

**Output**

**<class ‘float’>**

#### c. long

This Python Data type holds a long integer of unlimited length. But this construct does not exist in Python 3.x.

#### d. complex

This Python Data type holds a complex number. A complex number looks like this: a+bj Here, a and b are the real parts of the number, and j is imaginary.

>>> a=2+3j

>>> print(type(a))

**Output**

**<class ‘complex’>**

Use the isinstance() function to tell if Python variables belong to a particular class. It takes two parameters- the variable/value, and the class.

>>> print(isinstance(a,complex))

**Output**

**True**

2. **Strings**

A string is a sequence of characters. Python does not have a char data type, unlike C++ or Java. You can delimit a string using single quotes or double-quotes.

a="Hello"  
print(a)

## **String Concatenation**

To concatenate, or combine, two strings you can use the + operator. The operator + is used to concatenate two strings as the operation "Hello"+" Wolrd" returns "Hello World"

## a = "Hello" b = "World" c = a + b print(c)

a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)

## **String Multiplication**

The operator \* is known as a repetition operator as the operation "Hello" \*2 returns Hello Hello.

a="Hello"  
print(a\*2)

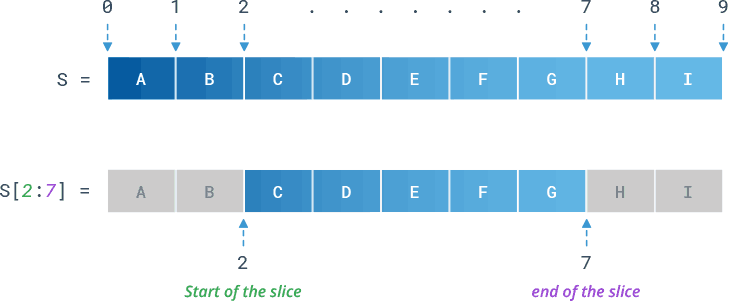
## **String Slicing**

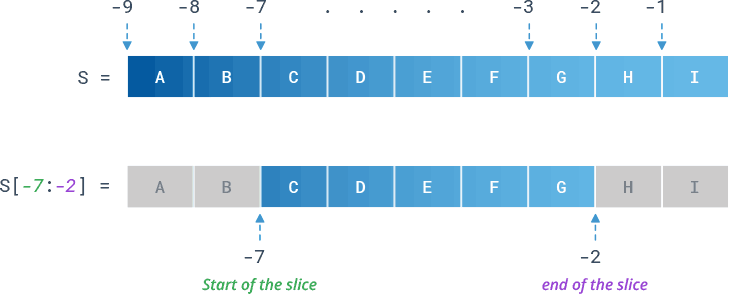
You can return a range of characters by using the slice syntax. Slicing is a technique for extracting parts of a string. It will display substring of a main string.

Specify the start index and the end index, separated by a colon, to return a part of the string.

In Python, index starts from 0.

**Index tracker for positive and negative index:** String indexing and slicing in python. Here, the Negative comes into consideration when tracking the string in reverse.





**Syntax**

String[start:stop] # items start through stop-1

String[start:] # items start through the rest of the String

String[:stop] # items from the beginning through stop-1

String[:] # a copy of the whole String

String[start:stop:step] # start through not past stop, by step

# string1= "CREDENCE"  
# print(string1)  
# print(string1[0])  
# print(string1[1])  
# print(string1[2])  
# print(string1[7])

# print(string1[8])

String = 'CredenceAutomation'

# print(String[0])  
# print(String[3])  
# print(String[0:7])  
# print(String[0:8])

# print(String[-1])  
# print(String[-3])  
# print(String[3:-10])  
# print(String[3:-16])  
# print(String[-3:1])  
# print(String[-3:18])  
# print(String[-2:-5])  
# print(String[-5:-2])

# print(String[0:])  
# print(String[8:])  
# print(String[:1])  
# print(String[:8])  
  
# print(String[1:10:2])  
# print(String[-1:-12:-2])  
  
# Prints string in reverse  
# print(String[::-1])

# print(String[::-4])

# Copy string to another variable

# copy = String[:]  
# print(copy)

# copy = String[0:2]  
# print(copy)  
# copy = String[2:7]  
# print(copy)

# copy = str(String)  
# print(copy)

# copy = '' + String  
# print(copy)

String[1]**=** "D"

As Strings are immutable in Python, if we try to update the string, then it will generate an error.

**String Methods**

1. **capitalize()**

**Python String capitalize()** method returns a copy of the original string and converts the first character of the string to a capital **(uppercase)** letter, while making all other characters in the string **lowercase**letters.

*# print(text.capitalize())*

2. casefold()

**Python String casefold()** method is used to convert string to lowercase. It is similar to the Python lower() string method.

*# print(text.casefold())*

*a="ßABc"  
# print(a.casefold())*

Avc

3. count()

Python String**count()** function is an inbuilt function in python programming language that returns the number of occurrences of a substring in the given string.

**Syntax:**

string.count(substring, start=…, end=…)

The count() function has one compulsory and two optional parameters.

* + **Mandatory parameter:**
    - substring – string whose count is to be found.
  + **Optional Parameters:**
    - start (Optional) – starting index within the string where the search starts.
    - end (Optional) – ending index within the string where the search ends.

*# print(text.count("e"))  
# print(text.count("e", 0, 5))  
# print(text.count("e", 0, 15))*

4. endswith()

**Python String endswith() method** returns True if a string ends with the given suffix, otherwise returns False.

print(text.endswith("TesTiNG"))

5. find()

**Python String find() method** returns the lowest index or first occurrence of the substring if it is found in a given string. If it is not found, then it returns -1.

**Syntax:**str\_obj.find(sub, start, end)

**Parameters:**

* **sub:** Substring that needs to be searched in the given string.
* **start (optional):** Starting position where the substring needs to be checked within the string.
* **end (optional):** End position is the index of the last value for the specified range. It is excluded while checking.

**Return:**  Returns the lowest index of the substring if it is found in a given string. If it’s not found then it returns -1.

*# print(text.find('A'))  
# print(text.find('Aut'))  
# print(text.find('Autom'))*

*# print(text.find('Automation'))*

*# print(text.find('A',0))  
# print(text.find('A',11))  
# print(text.find('A',2,3))*

6. Isinstance ()

Use the isinstance() function to tell if Python variables belong to a particular class. It takes two parameters- the variable/value, and the class.

>>> a=2+3j

>>> print(type(a))

>>> print(isinstance(a,complex))

7. index()

**Python String index()** Method allows a user to find the index of the first occurrence of an existing substring inside a given string.

Syntax- index(sub, start, end])

**Parameters**

* **sub** : substring
* **start** : start index a range
* **end** : last index of the range

*# print(String.index('A'))  
# print(String.index('A',1))  
# print(String.index('A',9))  
# print(String.index('A',10))*

*# print(String.index('A',10,30))*

Note- Both index() and find() are identical in that they return the index position of the first occurrence of the substring from the main string. The main difference is that Python find() produces -1 as output if it is unable to find the substring, whereas index() throws a ValueError exception.

8. isalnum()

**Python String isalnum()** method checks whether all the characters in a given string are either alphabet or numeric (alphanumeric) characters.

***Syntax:****string\_name.isalnum()*

***Parameter:****isalnum() method takes no parameters*

***Return:***

* ***True:****If all the characters are alphanumeric*
* ***False:****If one or more characters are not alphanumeric*

String1="Credence"  
print(String1.isalnum())-- True

String1="123ACS"  
print(String1.isalnum())-- True

String1 = 'Credence Automation 123 TesTiNG'

print(String.isalnum())—false becz of space

String1 = 'Credence\_Automation\_TesTiNG'

print(String.isalnum())— false becz of \_

String1 = 'Credence\_Automation\_TesTiNG'

String1="Credence"  
print(String1.isalnum())

9. isalpha()

**Python String** **isalpha()** method is used to check whether all characters in the String is an alphabet.

***Syntax:****string.isalpha()*

***Parameters:****isalpha() does not take any parameters*

***Returns:***

* ***True****: If all characters in the string are alphabet.*
* ***False****: If the string contains 1 or more non-alphabets.*
* *# String = 'Credence\_Automation\_TesTiNG'  
  # String1="Credence"  
  # print(String.isalpha())  
  # print(String1.isalpha())*

10. isdecimal()

**Python String isdecimal() function** returns true if all characters in a string are decimal, else it returns False.

**Syntax:** string\_name.isdecimal(), string\_name is the string whose characters are to be checked

**Parameters:**This method does not takes any parameters .

**Return:**boolean value.True – all characters are decimal, False – one or more than one character is not decimal.

s = "12345"  
print(s.isdecimal())  
  
*# contains alphabets*s = "12credence34"  
print(s.isdecimal())  
  
*# contains numbers and spaces*s = "12 34"  
print(s.isdecimal())

11. isnumeric()

Python String isnumeric() method returns “**True**” if all characters in the string are numeric characters, otherwise returns “**False**”.

***Syntax:****string.isnumeric()*

***Parameters:****isnumeric() does not take any parameters*

***Returns :***

* *True – If all characters in the string are numeric characters.*
* *False – If the string contains 1 or more non-numeric characters.*
* s = "12345"  
  print(s.isnumeric())  
    
  *# contains alphabets*s = "12credence34"  
  print(s.isnumeric())  
    
  *# contains numbers and spaces*s = "12 34"  
  print(s.isnumeric())

12. isdigit()

**Python String isdigit() method** returns “**True**” if all characters in the string are digits, Otherwise, It returns “False”.

**Python String isdigit() Method Syntax**

***Syntax:****string.isdigit()*

***Parameters:****isdigit() does not take any parameters*

***Returns:***

* *True – If all characters in the string are digits.*
* *False – If the string contains 1 or more non-digits.*

s = "12345"  
print(s.isdigit())  
  
*# contains alphabets*s = "12credence34"  
print(s.isdigit())  
  
*# contains numbers and spaces*s = "12 34"  
print(s.isdigit())

The differences between the these 3 methods

| **String Type** | **Example** | **Python .isdecimal()** | **Python .isdigit()** | **Python .isnumeric()** |
| --- | --- | --- | --- | --- |
| Base 10 Numbers | '0123' | True | True | True |
| Fractions and Superscripts | '⅔', '2²' | False | True | True |
| Roman Numerals | 'ↁ' | False | False | True |

s = "0123"  
print(s.isnumeric())  
  
*# contains alphabets*s = "0123"  
print(s.isdigit())  
  
*# contains numbers and spaces*s = "0123"  
print(s.isdecimal())

s = "2²"  
print(s.isnumeric())  
  
*# contains alphabets*s = "2²"  
print(s.isdigit())  
  
*# contains numbers and spaces*s = "2²"  
print(s.isdecimal())

s = "ↁ"  
print(s.isnumeric())  
  
*# contains alphabets*s = "ↁ"  
print(s.isdigit())  
  
*# contains numbers and spaces*s = "ↁ"  
print(s.isdecimal())

**13. islower()**

**Python String islower()** method checks if all characters in the string are lowercase.

This method returns **True**if all alphabets in a string are lowercase alphabets. If the string contains at least one uppercase alphabet, it returns **False**.

## Python String islower() Method Syntax

***Syntax:****string.islower()*

***Returns:***

* *True: If all the letters in the string are in lower case and*
* *False: If even one of them is in upper case.*

*# a="yusuf"  
# print(a.islower())  
# a="yuSuf"  
# print(a.islower())*

**14. istitle()**

**Python String istitle() Method**is a built-in string function that returns True if all the words in the string are title cased, otherwise returns False.

## Python String istitle() Method Syntax

***Syntax:****string.istitle()*

***Returns:****True if the string is a title-cased string otherwise returns False.*

When all words in a string begin with uppercase letters and the remaining characters are lowercase letters, the string is called title-cased.  This function ignores digits and special characters.

*# First character in each word is  
# uppercase and remaining lowercase*s = 'Credence For Credence'  
print(s.istitle())  
  
*# First character in first  
# word is lowercase*s = 'credence For Credence'  
print(s.istitle())  
  
*# Third word has uppercase  
# characters at middle*s = 'Credence For CREDENCE'  
print(s.istitle())  
*# Ignore the digit 6041, hence returns True*s = '6041 Is My Number'  
print(s.istitle())  
  
*# word has uppercase  
# characters at middle*s = 'CREDENCE'  
print(s.istitle())

**15. isupper()**

**Python String isupper()** method returns whether all characters in a string are uppercase or not.

## Python String isupper() method Syntax

***Syntax:****string.isupper()*

***Returns:****True if all the letters in the string are in the upper case and False if even one of them is in the lower case.*

*First character in each word is  
# uppercase and remaining lowercase*s = 'Credence For Credence'  
print(s.isupper())  
  
*# First character in first  
# word is lowercase*s = 'credence For Credence'  
print(s.isupper())  
  
*# Third word has uppercase  
# characters at middle*s = 'Credence For CREDENCE'  
print(s.isupper())  
*# Ignore the digit 6041, hence returns True*s = '6041 Is My Number'  
print(s.isupper())  
  
*# word has uppercase  
# characters at middle*s = 'CREDENCE'  
print(s.isupper())

**16. lower()**

Python String **lower()** method converts all uppercase characters in a string into lowercase characters and returns it.

**Syntax:** string.lower()

**Parameters:** The lower() method doesn’t take any parameters.

**Returns:** Returns a lowercase string of the given string

*# First character in first  
# word is lowercase*s = 'credence For Credence'  
print(s.lower())  
  
*# Third word has uppercase  
# characters at middle*s = 'Credence For CREDENCE'  
print(s.lower())  
*# Ignore the digit 6041, hence returns True*s = '6041 Is My Number'  
print(s.lower())  
  
*# word has uppercase  
# characters at middle*s = 'CREDENCE'  
print(s.lower())

**16. upper()**

Python String **upper()** method converts all lowercase characters in a string into uppercase characters and returns it.

**Syntax:** string.upper()

**Parameters:** The upper() method doesn’t take any parameters.

**Returns:** Returns a uppercase string of the given string

*# First character in each word is  
# uppercase and remaining uppercase*s = 'Credence For Credence'  
print(s.upper())  
  
*# First character in first  
# word is uppercase*s = 'credence For Credence'  
print(s.upper())  
  
*# Third word has uppercase  
# characters at middle*s = 'Credence For CREDENCE'  
print(s.upper())  
*# Ignore the digit 6041, hence returns True*s = '6041 Is My Number'  
print(s.upper())  
  
*# word has uppercase  
# characters at middle*s = 'CREDENCE'  
print(s.upper())

**17. title()**

**The String title() method** in Python is used to convert the first character in each word to uppercase and the remaining characters to lowercase in the string and returns a new string.

## Python String title() Method Syntax:

***Syntax:****str.title()*

***parameters:****title() doesn’t accept any parameter.*

***Return:****str, converted to title case*

*# First character in each word is  
# titlecase and remaining titlecase*s = 'Credence For Credence'  
print(s.title())  
  
*# First character in first  
# word is titlecase*s = 'credence For Credence'  
print(s.title())  
  
*# Third word has titlecase  
# characters at middle*s = 'Credence For CREDENCE'  
print(s.title())  
*# Ignore the digit 6041, hence returns True*s = '6041 Is My Number'  
print(s.title())  
  
*# word has titlecase  
# characters at middle*s = 'CREDENCE'  
print(s.title())

**18. len()**

**Python len() function** is an inbuilt function in Python. It can be used to find the length of an object.

### Python len() function Syntax:

**len(Object)**

**Parameter:**

* **Object:**Object of which we have to find the length for example string, list, etc.

**Returns:** It returns the integer value which indicates the length of an object.

s = 'Credence For Credence'  
print(len(s))  
  
  
s = 'credence For Credence'  
print(len(s))  
  
s = 'Credence For CREDENCE'  
print(len(s))  
  
s = '6041 Is My Number'  
print(len(s))  
  
  
s = 'CREDENCE'  
print(len(s))

19.replace()

**String replace()** in Python returns a copy of the string where occurrences of a substring are replaced with another substring.

## **Syntax of String replace() method**

The replace() method in Python strings has the following syntax:

***Syntax:****string.replace(old, new, count)*

***Parameters:***

* ***old –****old substring you want to replace.*
* ***new –****new substring which would replace the old substring.*
* ***count –****(****Optional****) the number of times you want to replace the old substring with the new substring.*

***Return Value :****It returns a copy of the string where all occurrences of a substring are replaced with another substring.*

Examples of Python replace() Method

Replace all Instances of a single character using replace() in Python

In this example, we are only replacing a single character from a given string. The Python replace() method is case-sensitive, and therefore it performs a case-sensitive substring substitution, i.e. R in FOR is unchanged.

string = "grrks FOR grrks"

# replace all instances of 'r' (old) with 'e' (new)

new\_string = string.replace("r", "e" )

print(string)

print(new\_string)

### Replace all Instances of a String using replace() in Python

Here, we will replace all the geeks with GeeksforGeeks using replace() function.

string = "geeks for geeks \ngeeks for geeks"

print(string)

# Prints the string by replacing only

# 3 occurrence of Geeks

print(string.replace("geeks", "GeeksforGeeks"))

### Replace only a certain number of Instances using replace() in Python

In this example, we are replacing certain numbers of words. i.e. “ek” with “a” with **count=3**.

string = "geeks for geeks geeks geeks geeks"

# Prints the string by replacing

# e by a

print(string.replace("e", "a"))

# Prints the string by replacing only

# 3 occurrence of ek by a

print(string.replace("ek", "a", 3))

20. swapcase()

Python String swapcase() method converts all uppercase characters to lowercase and vice versa of the given string and returns it.

***Syntax:***

*string\_name.swapcase()*

***Parameter:***

*The swapcase() method does not take any parameter.*

***Return Value:***

*The swapcase() method returns a string with all the cases changed.*

# Python program to demonstrate the use of

# swapcase() method

string = "gEEksFORgeeks"

# prints after swapping all cases

print(string.swapcase())

string = "geeksforgeeks"

print(string.swapcase())

string = "GEEKSFORGEEKS"

print(string.swapcase())

21.join()

**join()** is an inbuilt string function in Python used to join elements of the sequence separated by a string separator. This function joins elements of a sequence and makes it a string.

***Syntax:****string\_name.join(iterable)*

***Parameters:***

* *Iterable – objects capable of returning their members one at a time. Some examples are****List, Tuple, String, Dictionary****,****and Set***

***Return Value:****The join() method returns a string concatenated with the elements of iterable.*

String = "Credence"  
print("$".join(String))  
print(",".join(String))  
  
s1 = 'abc'  
s2 = '123'  
  
*# each element of s2 is separated by s1  
# '1'+ 'abc'+ '2'+ 'abc'+ '3'*print('s1.join(s2):', s1.join(s2))  
  
*# each element of s1 is separated by s2  
# 'a'+ '123'+ 'b'+ '123'+ 'b'*print('s2.join(s1):', s2.join(s1))

2. L**ist**

A list in Python is used to store the sequence of various types of data. A list can be defined as a collection of values or items of different types. Python lists are mutable type which implies that we may modify its element after it has been formed. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

Although Python has six data types that may hold sequences, the list is the most popular and dependable form. The collection of data is stored in a list, a sequence data type. Similar sequence data formats are Tuples and String.

Python lists are identical to dynamically scaled arrays that are specified in other languages, such as Java's ArrayList and C++'s vector. A list is a group of items that are denoted by the symbol [] and subdivided by commas.

**Characteristics of Lists**

The list has the following characteristics:

* The lists are ordered.
* The element of the list can access by index.
* The lists are the mutable type.
* The lists are allow to accept duplicates.
* A list can store the number of various elements.

**List Declaration**

# a simple list

list1 = [1, 2, "Python", "Program", 15.9]

list2 = ["Amy", "Ryan", "Henry", "Emma"]

# printing the list

**print**(list1)

**print**(list2)

# printing the type of list

**print**(type(list1))

**print**(type(list2))

**Python List Operations**

The concatenation (+) and repetition (\*) operators work in the same way as they were working with the strings. The different operations of list are

1. Repetition
2. Concatenation
3. Length
4. Membership
5. Iteration

**1. Repetition**

The repetition operator enables the list elements to be repeated multiple times.

**Code**

# repetition of list

# declaring the list

list1 = [12, 14, 16, 18, 20]

# repetition operator \*

l = list1 \* 2

**print**(l)

### 2. Concatenation

It concatenates the list mentioned on either side of the operator.

**Code**

# concatenation of two lists

# declaring the lists

list1 = [12, 14, 16, 18, 20]

list2 = [9, 10, 32, 54, 86]

# concatenation operator +

l = list1 + list2

**print**(l)

**Output:**

[12, 14, 16, 18, 20, 9, 10, 32, 54, 86]

### 3. Length

It is used to get the length of the list

**Code**

# size of the list

# declaring the list

list1 = [12, 14, 16, 18, 20, 23, 27, 39, 40]

# finding length of the list

len(list1)

**Output:**

9

### 4. Iteration

The for loop is used to iterate over the list elements.

**Code**

# iteration of the list

# declaring the list

list1 = [12, 14, 16, 39, 40]

# iterating

**for** i **in** list1:

**print**(i)

**Output:**

12

14

16

39

40

### 5. Membership

It returns true if a particular item exists in a particular list otherwise false.

**Code**

# membership of the list

# declaring the list

list1 = [100, 200, 300, 400, 500]

# true will be printed if value exists

# and false if not

**print**(600 **in** list1)

**print**(700 **in** list1)

**print**(1040 **in** list1)

**print**(300 **in** list1)

**print**(100 **in** list1)

**print**(500 **in** list1)

**Output:**

False

False

False

True

True

True

**Output:**

[12, 14, 16, 18, 20, 12, 14, 16, 18, 20]

**Ordered List Checking**

**Code**

1. # example
2. a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]
3. b = [ 1, 2, 5, "Ram", 3.50, "Rahul", 6 ]
4. a == b

**Output:**

False

The identical elements were included in both lists, but the second list modified the index position of the fifth element, which is against the lists' intended order. When the two lists are compared, false is returned.

**Code**

1. # example
2. a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6]
3. b = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6]
4. a == b

**Output:**

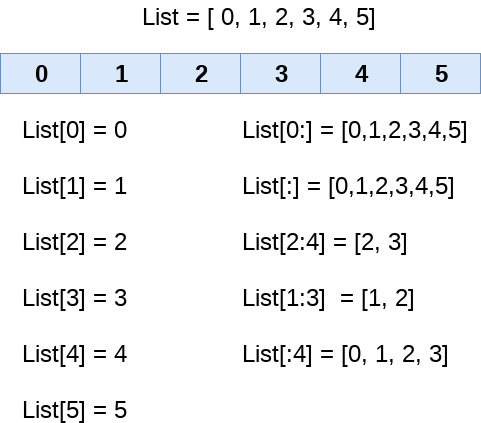
True

Lists permanently preserve the element's order. It is the arranged gathering of things because of this.

**List Indexing and Splitting**

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



We can get the sub-list of the list using the following syntax.

1. list\_varible(start:stop:step)

* The **start** denotes the starting index position of the list.
* The **stop** denotes the last index position of the list.
* The **step** is used to skip the nth element within a **start:stop**

The initial index is represented by the start parameter, the ending index is determined by the step, and also the number of elements which are "stepped" through is the value of the end parameter. In the absence of a specific value for step, the default value equals 1. Inside the resultant SubList, the item also with index start would be present, but the one with the index finish will not. A list's initial element seems to have the index of 0.

list = [1,2,3,4,5,6,7]

**print**(list[0])

**print**(list[1])

**print**(list[2])

**print**(list[3])

## Get all the Items

my\_list = [1, 2, 3, 4, 5]

print(my\_list[:])

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

**Output**

[1, 2, 3, 4, 5]

If you simply use :, you will get all the elements of the list. This is similar to print(my\_list).

## Get all the Items After a Specific Position

my\_list = [1, 2, 3, 4, 5]

print(my\_list[2:])

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

**Output**

[3, 4, 5]

If you want to get all the elements after a specific index, you can mention that index before : as shown in example above.

In the above example, elements at index 2 and all the elements after index 2 are printed.

**Note:** indexing starts from 0. Item on index 2 is also included.

## Get all the Items Before a Specific Position

my\_list = [1, 2, 3, 4, 5]

print(my\_list[:2])

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

**Output**

[1, 2]

This example lets you get all the elements before a specific index. Mention that index after :.

In the example, the items before index 2 are sliced. Item on index 2 is excluded.

## Get all the Items from One Position to Another Position

my\_list = [1, 2, 3, 4, 5]

print(my\_list[2:4])

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

**Output**

[3, 4]

If you want to get all the elements between two specific indices, you can mention them before and after :.

In the above example, my\_list[2:4] gives the elements between 2nd and the 4th positions. The starting position (i.e. 2) is included and the ending position (i.e. 4) is excluded.

## Get the Items at Specified Intervals

my\_list = [1, 2, 3, 4, 5]

print(my\_list[::2])

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

**Output**

[1, 3, 5]

If you want to get elements at specified intervals, you can do it by using two :.

In the above example, the items at interval 2 starting from index 0 are sliced.

If you want the indexing to start from the last item, you can use negative sign **-**.

my\_list = [1, 2, 3, 4, 5]

print(my\_list[::-2])

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

**Output**

[5, 3, 1]

The items at interval 2 starting from the last index are sliced.

If you want the items from one position to another, you can mention them from start to stop.

my\_list = [1, 2, 3, 4, 5]

print(my\_list[1:4:2])

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

**Output**

[2, 4]

The items from index 1 to 4 are sliced with intervals of 2.

# Slicing the elements

**print**(list[0:6])

# By default, the index value is 0 so its starts from the 0th element and go for index -1.

**print**(list[:])

**print**(list[2:5])

**print**(list[1:6:2])

**Output:**

1

2

3

4

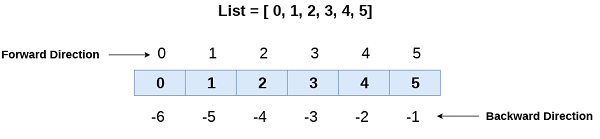
[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7]

[3, 4, 5]

[2, 4, 6]

In contrast to other languages, Python gives you the option to employ negative indexing as well. From the right, the negative indices are counted. The final element on the right-hand side of the list is represented by the index -1, followed by the next member on the left at the index -2, and so on until the last element on the left is reached.



1. # negative indexing example
2. list = [1,2,3,4,5]
3. **print**(list[-1])
4. **print**(list[-3:])
5. **print**(list[:-1])
6. **print**(list[-3:-1])

**Output:**

5

[3, 4, 5]

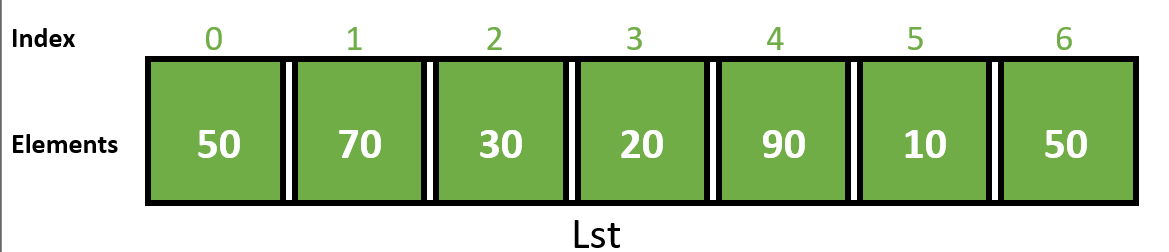
[1, 2, 3, 4]

[3, 4]

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

### Indexing

**1. Positive Indexes**



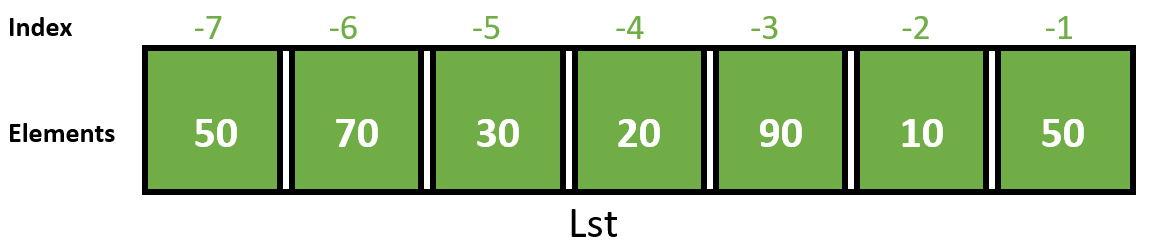
Lst **=** [50, 70, 30, 20, 90, 10, 50]

# Display list

print(Lst[::])

**2. Negative Indexes**

Now, let us look at the below diagram which illustrates a list along with its negative indexes.



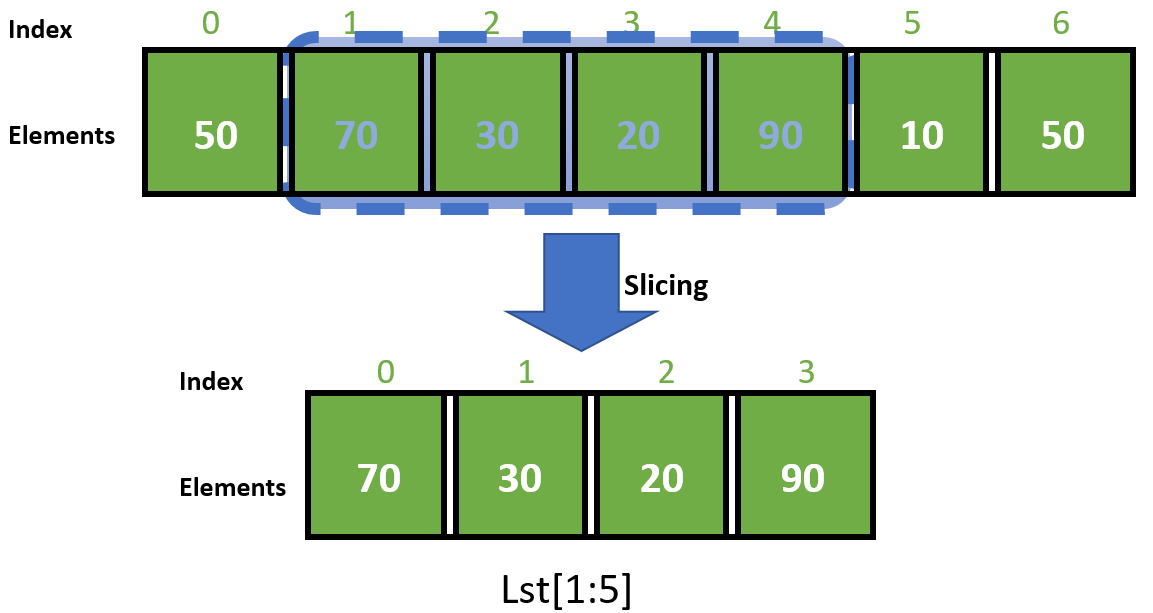
Lst **=** [50, 70, 30, 20, 90, 10, 50]

# Display list

print(Lst[**-**7::1])

**Slicing**

As mentioned earlier list slicing is a common practice in Python and can be used both with positive indexes as well as negative indexes. The below diagram illustrates the technique of list slicing:



The below program transforms the above illustration into python code:

* Python3

|  |
| --- |
| # Initialize list  Lst **=** [50, 70, 30, 20, 90, 10, 50]    # Display list  print(Lst[1:5]) |

**Output:**

[70, 30, 20, 90]

Below are some examples which depict the use of list slicing in Python:

**Example 1:**

* Python3

|  |
| --- |
| # Initialize list  List **=** [1, 2, 3, 4, 5, 6, 7, 8, 9]    # Show original list  **print**("\nOriginal List:\n", List)    print("\nSliced Lists: ")    # Display sliced list  **print**(List[3:9:2])    # Display sliced list  print(List[::2])    # Display sliced list  print(List[::]) |

**Output:**

Original List:

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

Sliced Lists:

[4, 6, 8]

[1, 3, 5, 7, 9]

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

Leaving any argument like *Initial*, *End* or *IndexJump* blank will lead to the use of default values i.e 0 as *Initial*, length of list as *End* and 1 as *IndexJump*.

**Example 2:**

* Python3

|  |
| --- |
| # Initialize list  List **=** ['Geeks', 4, 'geeks !']    # Show original list  print("\nOriginal List:\n", List)    print("\nSliced Lists: ")    # Display sliced list  print(List[::**-**1])    # Display sliced list  **print**(List[::**-**3])    # Display sliced list  **print**(List[:1:**-**2]) |

**Output:**

Original List:

['Geeks', 4, 'geeks !']

Sliced Lists:

['geeks !', 4, 'Geeks']

['geeks !']

['geeks !']

A reversed list can be generated by using a negative integer as the *IndexJump*argument*.*Leaving the *Initial*and *End*as blank*.*We need to choose the *Initial*and*End*value according to a reversed list if the *IndexJump*value is negative.

**Example 3:**

* Python3

|  |
| --- |
| # Initialize list  List **=** [**-**999, 'G4G', 1706256, '^\_^', 3.1496]    # Show original list  **print**("\nOriginal List:\n", List)    **print**("\nSliced Lists: ")    # Display sliced list  **print**(List[10::2])    # Display sliced list  **print**(List[1:1:1])    # Display sliced list  print(List[**-**1:**-**1:**-**1])    # Display sliced list  print(List[:0:]) |

**Output:**

Original List:

[-999, 'G4G', 1706256, '^\_^', 3.1496]

Sliced Lists:

[]

[]

[]

[]

If some slicing expressions are made that do not make sense or are incomputable then empty lists are generated.

**Example 4:**

* Python3

|  |
| --- |
| # Initialize list  List **=** [**-**999, 'G4G', 1706256, 3.1496, '^\_^']    # Show original list  **print**("\nOriginal List:\n", List)      print("\nSliced Lists: ")    # Modified List  List[2:4] **=** ['Geeks', 'for', 'Geeks', '!']    # Display sliced list  **print**(List)    # Modified List  List[:6] **=** []    # Display sliced list  print(List) |

**Output:**

Original List:

[-999, 'G4G', 1706256, 3.1496, '^\_^']

Sliced Lists:

[-999, 'G4G', 'Geeks', 'for', 'Geeks', '!', '^\_^']

['^\_^']

List slicing can be used to modify lists or even delete elements from a list.

**Example 5:**

* Python3

|  |
| --- |
| # Initialize list  List **=** [1, 2, 3, 4, 5, 6, 7, 8, 9]    # Show original list  **print**("\nOriginal List:\n", List)    print("\nSliced Lists: ")    # Creating new List  newList **=** List[:3]**+**List[7:]    # Display sliced list  print(newList)    # Changing existing List  List **=** List[::2]**+**List[1::2]    # Display sliced list  print(List) |

**Output:**

Original List:

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

Sliced Lists:

[1, 2, 3, 8, 9]

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

By concatenating sliced lists, a new list can be created or even a pre-existing list can be modified.

**Updating List Values**

Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the slice and assignment operator. Python also provides append() and insert() methods, which can be used to add values to the list.

1. # updating list values
2. list = [1, 2, 3, 4, 5, 6]
3. **print**(list)
4. # It will assign value to the value to the second index
5. list[2] = 10
6. **print**(list)
7. # Adding multiple-element
8. list[1:3] = [89, 78]
9. **print**(list)
10. # It will add value at the end of the list
11. list[-1] = 25
12. **print**(list)

**Output:**

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

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]

To change the value of a specific item, refer to the index number:

Change the second item

thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]  
thislist[1:3] = ["blackcurrant", "watermelon"]  
print(thislist)

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Change the second value by replacing it with two new values:

thislist = ["apple", "banana", "cherry"]  
thislist[1:2] = ["blackcurrant", "watermelon"]  
print(thislist)

If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

### Example

Change the second and third value by replacing it with one value:

thislist = ["apple", "banana", "cherry"]  
thislist[1:3] = ["watermelon"]  
print(thislist)

**3. Tuple**

A tuple is an ordered collection of values.

Tuples are a lot like lists:

* **Tuples are ordered** – Tuples maintains a left-to-right positional ordering among the items they contain.
* **Accessed by index** – Items in a tuple can be accessed using an index.
* **Tuples can contain any sort of object** – It can be numbers, strings, lists and even other tuples.
* Allow Duplicates

except:

* **Tuples are immutable** – you can’t add, delete, or change items after the tuple is defined.
* Tuples are written with round brackets where as Lists will be in Square Brackets

## Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, [string](https://www.programiz.com/python-programming/string), etc.).

# Different types of tuples

# Empty tuple

my\_tuple = ()

print(my\_tuple)

# Tuple having integers

my\_tuple = (1, 2, 3)

print(my\_tuple)

# tuple with mixed datatypes

my\_tuple = (1, "Hello", 3.4)

print(my\_tuple)

# nested tuple

my\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

print(my\_tuple)

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

**Output**

()

(1, 2, 3)

(1, 'Hello', 3.4)

('mouse', [8, 4, 6], (1, 2, 3))

In the above example, we have created different types of tuples and stored different data items inside them.

As mentioned earlier, we can also create tuples without using parentheses: But we mostly use ().

my\_tuple = 1, 2, 3

my\_tuple = 1, "Hello", 3.4

## Create a Python Tuple With one Element

In Python, creating a tuple with one element is a bit tricky. Having one element within parentheses is not enough.

We will need a trailing comma to indicate that it is a tuple,

var1 = ("Hello") # string

var2 = ("Hello",) # tuple

We can use the type() function to know which class a variable or a value belongs to.

var1 = ("hello")

print(type(var1)) # <class 'str'>

# Creating a tuple having one element

var2 = ("hello",)

print(type(var2)) # <class 'tuple'>

# Parentheses is optional

var3 = "hello",

print(type(var3)) # <class 'tuple'>

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

Here,

* ("hello") is a string so type() returns str as class of var1 i.e. <class 'str'>
* ("hello",) and "hello", both are tuples so type() returns tuple as class of var1 i.e. <class 'tuple'>

## **Tuple Concatenation,Repetition,Membership,Iteration and Length**

Tuples can be joined using the concatenation operator + or Replication operator \*

# Concatenate

tuple1 = (101, 102, 103) + (104, 105, 106)  
print(tuple1)  
*# Prints (101, 102, 103, 104, 105, 106)*

T = ('red', 'green', 'blue') + (1, 2, 3)  
print(T)  
*# Prints ('red', 'green', 'blue', 1, 2, 3)*print(tuple1+T)  
*# Prints (101, 102, 103, 104, 105, 106, 'red', 'green', 'blue', 1, 2, 3)*tuple2=tuple1+T  
print(tuple2)  
*# Prints (101, 102, 103, 104, 105, 106, 'red', 'green', 'blue', 1, 2, 3)  
  
# Replicate(Repeatation / Multiplication)*T = ('red',) \* 3  
print(T)  
*# Prints ('red', 'red', 'red')*T = ('red', 'Green') \* 3  
print(T)  
*# Prints ('red', 'Green', 'red', 'Green', 'red', 'Green')*

*#Membership*

*T = ('red', 'green', 'blue')*

print("red" in T)  
  
#Iteration

for i in tup1:  
 print(i)  
  
#Length

tuple1 = (12, 14, 16, 18, 20, 23, 27, 39, 40)  
*# finding length of the tuple*print(len(tuple1))

**Change Tuple Items**

Tuples are immutable (unchangeable). Once a tuple is created, it cannot be modified.

T = ('red', 'green', 'blue')

T[0] = 'black'

# Triggers TypeError: 'tuple' object does not support item assignment

tuple1 **=** (0, 1, 2, 3)

tuple1[0] **=** 4

print(tuple1)

**Output:**

Traceback (most recent call last):

File "e0eaddff843a8695575daec34506f126.py", line 3, in

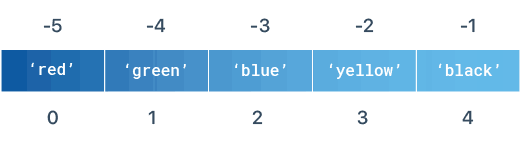
tuple1[0]=4

TypeError: 'tuple' object does not support item assignment

## **Accessing Values in Tuples**

You can access individual items in a tuple using an index in square brackets. Note that tuple indexing starts from 0.

The indices for the elements in a tuple are illustrated as below:



T = ('red', 'green', 'blue', 'yellow', 'black')

print(T[0])

# Prints red

print(T[2])

# Prints blue

You can access a tuple by negative indexing as well. Negative indexes count backward from the end of the tuple. So, T[-1] refers to the last item, T[-2] is the second-last, and so on

T = ('red', 'green', 'blue', 'yellow', 'black')

print(T[-1])

# Prints black

print(T[-2])

# Prints yellow

tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5, 6, 7 );  
print ("tup1[0]: ", tup1[0])  
print ("tup2[1:5]: ", tup2[1:5])

When the above code is executed, it produces the following result –

tup1[0]: physics

tup2[1:5]: (2, 3, 4, 5)

## **Tuple Slicing**

To access a range of items in a tuple, you need to slice a tuple using a slicing operator. Tuple slicing is similar to list slicing.

T = ('a', 'b', 'c', 'd', 'e', 'f')

print(T[2:5])

# Prints ('c', 'd', 'e')

print(T[0:2])

# Prints ('a', 'b')

print(T[3:-1])

# Prints ('d', 'e')

tuple1 **=** (0 ,1, 2, 3)

print(tuple1[1:])

**print**(tuple1[::**-**1])

print(tuple1[2:4])

**Output:**

(1, 2, 3)

(3, 2, 1, 0)

(2, 3)

my\_tuple = (c, 'r', e, ‘d’, 'e', 'n', 'c', 'e')

# elements beginning to end

print(my\_tuple[:]) # Prints (c, 'r', e, ‘d’, 'e', 'n', 'c', 'e')

## **Delete a Tuple**

Tuples cannot be modified, so obviously you cannot delete any item from it. However, you can delete the tuple completely with del keyword.

T = ('red', 'green', 'blue')

del T

*# tuple3 = ( 0, 1)  
# del tuple3  
# print(tuple3)*

t = ('red', 'green', 'blue')  
del(t[0])  
print(t)

TypeError: 'tuple' object doesn't support item deletion---- immutable

*# a = ['red', 'green', 'blue']  
# del a[0]  
# print(a)*

*List Supports deletion Updation ---Mutable*

## **Tuple Packing & Unpacking**

### Tuple Packing

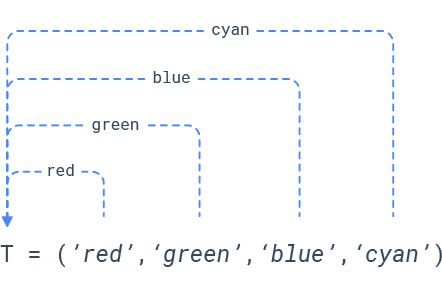
When a tuple is created, the items in the tuple are packed together into the object.

T = ('red', 'green', 'blue', 'cyan')

print(T)

# Prints ('red', 'green', 'blue', 'cyan')

In above example, the values ‘red’, ‘green’, ‘blue’ and ‘cyan’ are packed together in a tuple.



### Tuple Unpacking

When a packed tuple is assigned to a new tuple, the individual items are **unpacked** (assigned to the items of a new tuple).

T = ('red', 'green', 'blue', 'cyan')

(a, b, c, d) = T

print(a)

# Prints red

print(b)

# Prints green

print(c)

# Prints blue

print(d)

# Prints cyan

*# T = ('red', 'green', 'blue', 'cyan') #Packing  
# (a, b, c) = T # Unpacking  
# print(a) # ValueError: too many values to unpack (expected 3)  
  
# T = ('red', 'green', 'blue') #Packing  
# (a, b, c, d) = T # Unpacking  
# print(a) #* *not enough values to unpack (expected 4, got 3)*

**Python Tuple Methods**

Following are the tuple [methods](https://www.edureka.co/blog/python-functions) that we can use while working with a tuple in python.

* count(): Returns the count of the items.
* index(): It returns the index of the item specified.
* tuple()-Use tuple() to converts a data type to tuple. For example, in the code chunk below, you convert a Python list to a tuple.

|  |  |
| --- | --- |
| 1  2  3 | a **=** (1,2,1,3,1,3,1,2,1,4,1,5,1,5)  print(a.count(1))  print(a.index(5))  abc=("Yusuf","Amit","Pooja","raj", "Pritesh","Priya","Yusuf") print(abc.count("Yusuf")) print(abc.index("Yusuf")) print(abc.index("Yusuf",1,7)) print(abc.index("Yusuf",-7,-1)) print(abc.index("Yusuf",-1,-7)) |

a\_list = [1,2,3,4,5]

b\_tuple = tuple(a\_list)

## print(type(b\_tuple))

## The min(), max(), and sum() Tuple Functions

min(): gives the smallest element in the tuple as an output. Hence, the name is min().

For example,

max(): gives the largest element in the tuple as an output. Hence, the name is max().

For example,

COPY CODE

>>> tup = (22, 3, 45, 4, 2.4, 2, 56, 890, 1)

>>> max(tup)

890

max(): gives the sum of the elements present in the tuple as an output.

For example,

COPY CODE

>>> tup = (22, 3, 45, 4, 2, 56, 890, 1)

>>> sum(tup)

1023

tup2 = ("Yusuf","yusuf")  
  
print(max(tup2))  
print(min(tup2))

sorted()

To return a tuple with the elements in an sorted order, use sorted(), just like in the following example:

tuple\_ = (5, 2, 24, 3, 1, 6, 7)

sorted\_ = tuple(sorted(tuple\_))

print('Sorted Tuple :', sorted\_)

print(type(sorted\_))

**Output:**

Sorted Tuple : (1, 2, 3, 5, 6, 7, 24)

<class 'tuple'>

tuple\_ = (5, 2, 24, 3, 1, 6, 7)

sorted\_ = tuple(sorted(tuple\_, reverse=True))

print('Sorted Tuple :', sorted\_)

print(type(sorted\_))

**Output:**

Sorted Tuple : (24, 7, 6, 5, 3, 2, 1)

<class 'tuple'>

len()

With the len() function, you can returns the length of the tuple:

a = (1,2,3,4,5)

print(len(a))

**Type casting of string list and tuple**

We use str(),list(),tuple() respectively.

*# a\_list = [1,2,3,4,5]  
# print(a\_list)  
# print(type(a\_list)) # list  
# b\_tuple = tuple(a\_list)  
# print(b\_tuple)  
# print(type(b\_tuple)) #tuple  
#  
# print(a\_list)  
# print(type(a\_list))  
#  
# a\_list = tuple(a\_list)  
# print(a\_list)  
# print(type(a\_list))  
  
# a="Yusuf"  
# a=tuple(a)  
# print(a)  
# print(type(a))  
#   
# a="Yusuf"  
# a=list(a)  
# print(a)  
# print(type(a))  
  
# print(a)  
# a=str(a)  
# print(a)  
# print(type(a))*

## **Change Tuple Values with converting it into list**

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

### Example

Convert the tuple into a list to be able to change it:

x = ("apple", "banana", "cherry")  
y = list(x)  
y[1] = "kiwi"  
x = tuple(y)  
  
print(x)

### Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

thistuple = ("apple", "banana", "cherry")  
y = list(thistuple)  
y.append("orange")  
thistuple = tuple(y)

thistuple = ("apple", "banana", "cherry")  
y = ("orange",)  
thistuple += y  
  
print(thistuple)

tuple1 = (5, 3, 2, 8, 4, 4, 6, 2)

#change tuple to list

list1 = list(tuple1)

#update list

list1[2] = 63

#change back list to tuple

tuple1 = tuple(list1)

print(tuple1)

**Output**

(5, 3, 63, 8, 4, 4, 6, 2)

tuple1 = (5, 3, 2, 8, 4, 4, 6, 2)

#change tuple to list

list1 = list(tuple1)

#remove an item from list

list1.remove(2)

#change back list to tuple

tuple1 = tuple(list1)

print(tuple1)

**Output**

(5, 3, 8, 4, 4, 6, 2)

# Python Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

Set is a collection of well-defined objects as per mathematical definition. Set in python is the same as in mathematics. Set is a Build in Data Type in python to store different unique, iterable data types in the unordered form. Set can have elements of many data types such as int, string, tuple, etc.

Set is based on the hashing concept which makes it optimized in the search operation. So, it is used in case we have to perform many search operations on the collection of data. Set data structure is used when we want each element to be unique in a large collection of data.

x = {**1**,**2.3**, "py", (**1**,**2**,**3**)}

print(x)

## Creating a set

## By set() Constructor

A set can be created with a set() constructor by passing an iterable element such as list, string, etc in the method.

**Syntax:**

variable = set(iterable element)

### By Curly {} brackets:

A set can be created bypassing elements inside curly braces separated by a comma. These Elements can be any Immutable data type such as int, string, bool, tuple, etc but can not be List, sets, dictionaries.

**Syntax:**

variable = {element1, element2,..}

**Program:**

# Initialization of set by set() Constructor

x = set("python")

print('By set() constructor: ', x)

# Initialization of set by curly {}brackets

y={'y', 'n', 't', 'p', 'h', 'o'}

print('curly {}brackets: ', y)

### Example 1: Using curly braces

Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}

**print**(Days)

**print**(type(Days))

# create a set of integer type

student\_id = {112, 114, 116, 118, 115}

print('Student ID:', student\_id)

# create a set of string type

vowel\_letters = {'a', 'e', 'i', 'o', 'u'}

print('Vowel Letters:', vowel\_letters)

# create a set of mixed data types

mixed\_set = {'Hello', 101, -2, 'Bye'}

print('Set of mixed data types:', mixed\_set)

### Example 2: Using set() method (Constructor)

# create a set of integer type

student\_id = set[112, 114, 116, 118, 115]

print('Student ID:', student\_id)

# create a set of string type

vowel\_letters = set['a', 'e', 'i', 'o', 'u']

print('Vowel Letters:', vowel\_letters)

# create a set of mixed data types

mixed\_set = set['Hello', 101, -2, 'Bye']

print('Set of mixed data types:', mixed\_set)

Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])

**print**(Days)

**print**(type(Days))

# Initialization of set by set() Constructor

x = set("python")

print('By set() constructor: ', x)

# Initialization of set by curly {}brackets

y={'y', 'n', 't', 'p', 'h', 'o'}

print('curly {}brackets: ', y)

Create an Empty Set in Python

# create an empty set

empty\_set = set()

# create an empty dictionary

empty\_dictionary = {}

# check data type of empty\_set

print('Data type of empty\_set:', type(empty\_set))

# check data type of dictionary\_set

print('Data type of empty\_dictionary', type(empty\_dictionary))

## Properties of Set Data Type:

**Set follow three properties:**

1. **Unordered**
2. **Unique**
3. **Immutable**
4. Set store the element in an unordered manner such that no direct access to the element is possible. We use an iterator to iterate over all elements of the set.
5. Set store the element uniquely. No duplicate element is allowed. If we try to insert an element that is already present in the set then the element will not insert and remain as it is. The frequency of each element is set as one.
6. Each element in the set is immutable which means elements can not be changed. Hence sets can contain elements of Datatype int, string, Tuple, etc but can not contain List, sets, dictionaries. set elements are immutable but the set is mutable which means we can insert and delete elements from the set.

## Traversal the set or Iterate Over a Set in Python

Set can be traversed with help of (for in) keywords. As no direct access to the element is possible therefore only linear traversal is possible in the set.

**Syntax:**

for element in set:

print(element)

**Program:**

# Initialization of set x

x={'p','h','t','n','o'}

# Traversal the set with for loop

for y in x:

print(y)

fruits = {"Apple", "Peach", "Mango"}

# for loop to access each fruits

for fruit in fruits:

print(fruit)

## Duplicate Items in a Set

Let's see what will happen if we try to include duplicate items in a set.

numbers = {2, 4, 6, 6, 2, 8}

print(numbers) # {8, 2, 4, 6}

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

Here, we can see there are no duplicate items in the set as a set cannot contain duplicates.

## Add and Update Set Items in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

### Add Items to a Set in Python

In Python, we use the add() method to add an item to a set. For example,

numbers = {21, 34, 54, 12}

print('Initial Set:',numbers)

# using add() method

numbers.add(32)

print('Updated Set:', numbers)

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

**Output**

Initial Set: {34, 12, 21, 54}

Updated Set: {32, 34, 12, 21, 54}

In the above example, we have created a set named numbers. Notice the line,

numbers.add(32)

Here, add() adds **32** to our set.

### Update Python Set

The update() method is used to update the set with items other collection types (lists, tuples, sets, etc).

**Update()**method is used to add multiple elements into the set. Multiple elements can be in form of iterable data types such as set, list, tuple, string, etc

For example,

companies = {'Lacoste', 'Ralph Lauren'}

tech\_companies = ['apple', 'google', 'apple']

companies.update(tech\_companies)

print(companies)

# Output: {'google', 'apple', 'Lacoste', 'Ralph Lauren'}

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

Here, all the unique elements of tech\_companies are added to the companies set.

y={**2**,**3**,**4**,**5**}

print('Set y: ', y)

# Update y to set x

x.update(y)

print('Set y: ', y)

print('Set x after update: ', x)

## Deletion from the set

Deletion operation on the set can be performed by these four methods:

1. remove
2. discard
3. pop
4. clear

### Remove

Remove method removes an element from the set which is passed in the method from the set. If the element is not present in the set then it raises an error message.

**Syntax:**

set.remove(element)

### Discard

**Discard** method removes an element from the set which is passed in the method from the set. If the element is not present in the set then it gives no error message.

**Syntax:**

set.discard(element)

### pop()

The pop() method randomly removes an item from a set and returns the removed item.

**Syntax:**

variable = set.pop()

### clear()

**clear()**method removes all the elements of the set (set becomes null).

**Syntax:**

set.clear()

# Initialization of set x

x={'Python','Java','PHP','Angular'}

print('Set x: ', x)

x.remove('Java')

print('Set x after remove: ', x)

x.discard('PHP')

print('Set x after discard: ', x)

# Initialization set x

x={**1**,**2**,"py"}

print('Print set x: ', x)

# pop last element from the set x

z=x.pop()

print('Print first element of set x: ', z)

# clear all element from set

x.clear()

print('Print set x after clear: ', x)

languages = {'Python', 'Java', 'English'}

# remove English from the set

languages.remove('English')

print(languages)

# Output: {'Python', 'Java'}

numbers = {2, 3, 4, 5}

# discards 3 from the set

numbers.discard(3)

print('Set after discard:', numbers)

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

A = {'a', 'b', 'c', 'd'}

removed\_item = A.pop()

print(removed\_item)

# Output: c

# set of prime numbers

primeNumbers = {2, 3, 5, 7}

# clear all elements

primeNumbers.clear()

print(primeNumbers)

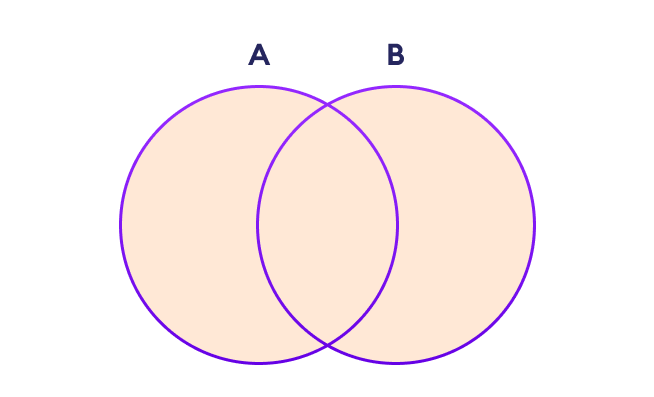
# set()

## **Python Set Operations**

Python Set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction, and symmetric difference.

### Union of Two Sets

The union of two sets **A** and **B** include all the elements of set **A** and **B and not include duplicates**.

Set Union in Python



We use the | operator or the union() method to perform the set union operation. For example,

# first set

A = {1, 3, 5,2}

# second set

B = {0, 2, 4,1}

# perform union operation using |

print('Union using |:', A | B)

# perform union operation using union()

print('Union using union():', A.union(B))

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

**Output**

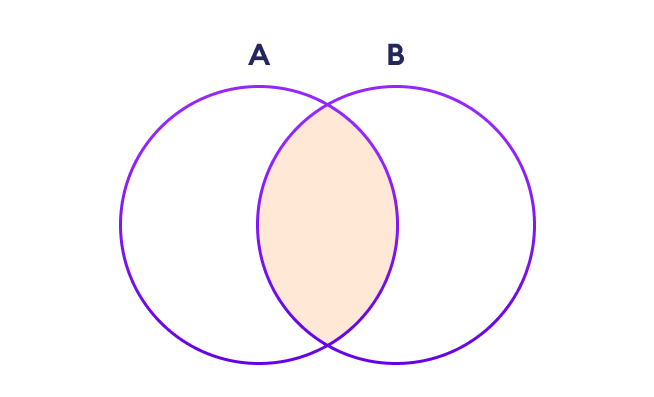
Union using |: {0, 1, 2, 3, 4, 5}

Union using union(): {0, 1, 2, 3, 4, 5}

**Note**: A|B and union() is equivalent to A ⋃ B set operation.

### Set Intersection

The intersection of two sets **A** and **B** include the common elements between set **A** and **B**.

Set Intersection in Python



In Python, we use the & operator or the intersection() method to perform the set intersection operation. For example,

# first set

A = {1, 3, 5}

# second set

B = {1, 2, 3}

# perform intersection operation using &

print('Intersection using &:', A & B)

# perform intersection operation using intersection()

print('Intersection using intersection():', A.intersection(B))

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

**Output**

Intersection using &: {1, 3}

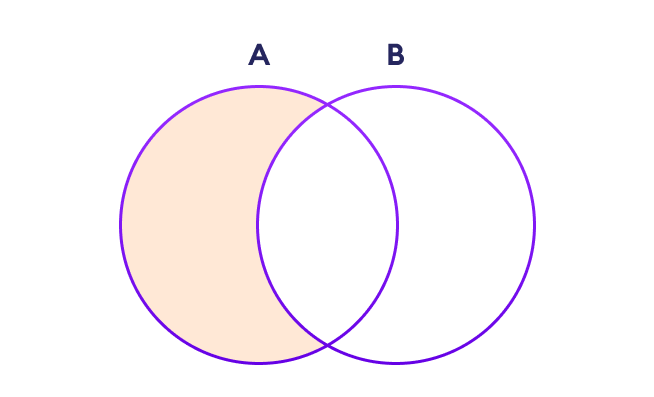
Intersection using intersection(): {1, 3}

**Note**: A&B and intersection() is equivalent to A ⋂ B set operation.

### Difference between Two Sets

The difference between two sets **A** and **B** include elements of set **i** that are not present on set **B**.



Set Difference in Python



We use the - operator or the difference() method to perform the difference between two sets. For example,

# first set

A = {2, 3, 5}

# second set

B = {1, 2, 6}

# perform difference operation using &

print('Difference using &:', A - B)

# perform difference operation using difference()

print('Difference using difference():', A.difference(B))

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

**Output**

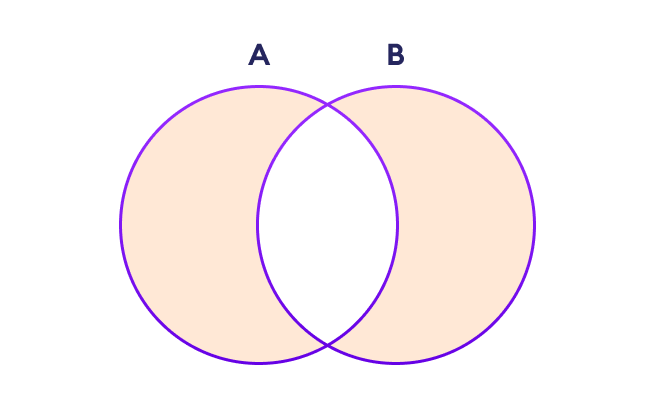
Difference using &: {3, 5}

Difference using difference(): {3, 5}

**Note**: A - B and A.difference(B) is equivalent to A - B set operation.

### Set Symmetric Difference

The symmetric difference between two sets **A** and **B** includes all elements of **A** and **B** without the common elements.

Set Symmetric Difference in Python



In Python, we use the ^ operator or the symmetric\_difference() method to perform symmetric difference between two sets. For example,

# first set

A = {2, 3, 5}

# second set

B = {1, 2, 6}

# perform difference operation using &

print('using ^:', A ^ B)

# using symmetric\_difference()

print('using symmetric\_difference():', A.symmetric\_difference(B))

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

**Output**

using ^: {1, 3, 5, 6}

using symmetric\_difference(): {1, 3, 5, 6}

[issubset](https://www.stechies.com/python-sets/#issubset)

Issubset checks if the first set is a subset of another set.

**Syntax:**

bool variable=set1.issubset(set2)

[isdisjoint](https://www.stechies.com/python-sets/#isdisjoint)

Isdisjoint checks if there is no common element between two sets.

**Syntax:**

bool variable=set1.isdisjoint(set2)

# Initialization of set x and y

x={**1**,**2**}

y={**1**,**2**,**3**,**4**}

# check if set x is subsite of y

z=x.issubset(y)

print('Check if x is subset of y: ', z)

# Initialization set x and y

x={**1**,**2**}

y={**3**,**4**}

# Check if set x and y are disjoint

z=x.isdisjoint(y)

print('Check if set x and y are disjoint:', z)

## Operator with set:

**There are many operators which can be used with sets . Some of them are as follow:**

|  |  |
| --- | --- |
| == | Checks if two sets are equal or have the same elements. |
| != | Checks if two sets are not equal or have different elements. |
| <= | Checks if the first set is a subset of another set. |
| < | Checks if the first set is a proper subset of another set. |
| >= | Checks if the first set is a superset of another set. |
| > | Checks if the first set is a proper superset of another set. |
| & | This operator takes the intersection of two sets. |
| | | This operator takes the union of two sets. |
| - | This operator takes the difference of two sets. |
| ^ | This operator takes the Symmetric Difference of two sets. |

**Program**

# Initialization of set x and y

x = {'a','b','c'}

y = {'a','b','c'}

print('Set x: ', x)

print('Set y: ', y)

# Operator with set

print('Set x == y: ', x==y)

print('Set x != y: ', x != y)

# Initialization of set x and y

x = {**1**,**2**}

y = {**1**,**2**,**3**,**4**}

print('Set x: ', x)

print('Set y: ', y)

print('Set x <= y: ', x <= y)

print('Set x < y: ', x < y)

# Initialization of set x and y

x = {**1**,**2**,**3**,**4**}

y = {**1**,**2**,**3**}

print('Set x superset y:', x >= y)

print('Set x proper superset y:', x > y)

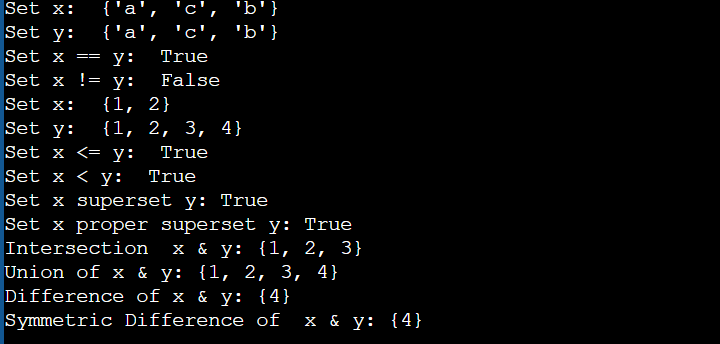
print('Intersection x & y:', x & y)

print('Union of x & y:', x | y)

print('Difference of x & y:', x - y)

print('Symmetric Difference of x & y:', x ^ y)

**Output:**



## Some Build in Methods:

There is some built in function in the set:

|  |  |
| --- | --- |
| Copy () | To shallow copy a set to another set. |
| in | Check if an element is in the set. |
| not in | Check if an element is not in the set. |
| len () | Find the number of elements in the set. |
| max () | Find the max element in the set. |
| min () | Find the minimum element in the set. |
| sorted () | Sort the elements of the set. |
| sum () | Find the sum of elements of the set. |
| all () | Check if all the iterable elements of the set are true. |
| any () | Check if any of the iterable elements of the set is true. |

**Program:**

*# Initialization of set x*x={20,1,2,3,4,10}  
  
*# copy set x to set z*z=x.copy()  
  
print('Copy set x to set z: ', z)  
  
print('Print length of set z: ',len(z) )  
  
  
print('Print min of set z: ',min(z))  
  
  
print('Print max of set z: ',max(z))  
  
print('Print Sum of set z: ',sum(z))  
  
  
*# Sort set z*print(sorted(z))  
print(sorted(z, reverse=True))  
print(all(x))  
print(any(x))

print(5 in x)  
print(5 not in x)

**Boolean Data Type**

**Python** **boolean** type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False. Generally, it is used to represent the truth values of the expressions. For example, 1==1 is True whereas 2<1 is False.

The boolean value can be of two types only i.e. either True or False. The output ***<class ‘bool’>*** indicates the variable is a boolean data type.

*a = True*

*print(type(a))*

*b = False*

*print(type(b))*

*# a=10  
# b=20  
# print(a>b)  
# print(type(a>b))  
#   
# a=10  
# b=20  
# print(a< b)  
# print(type(a<b))*

**Python frozenset()**

The frozenset() function returns an immutable frozenset object initialized with elements from the given iterable.

Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

Due to this, frozen sets can be used as keys in Dictionary or as elements of another set. But like sets, it is not ordered (the elements can be set at any index).

The syntax of frozenset() function is:

frozenset([iterable])

## frozenset() Parameters

The frozenset() function takes a single parameter:

* **iterable (Optional)** - the iterable which contains elements to initialize the frozenset with.  
  Iterable can be set, dictionary, tuple, etc.

## Return value from frozenset()

The frozenset() function returns an immutable frozenset initialized with elements from the given iterable.

If no parameters are passed, it returns an empty frozenset.

## Example 1: Working of Python frozenset()

# tuple of vowels

vowels = ('a', 'e', 'i', 'o', 'u')

fSet = frozenset(vowels)

print('The frozen set is:', fSet)

print('The empty frozen set is:', frozenset())

# frozensets are immutable

fSet.add('v')

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

**Output**

The frozen set is: frozenset({'a', 'o', 'u', 'i', 'e'})

The empty frozen set is: frozenset()

Traceback (most recent call last):

File "<string>, line 8, in <module>

fSet.add('v')

AttributeError: 'frozenset' object has no attribute 'add'

## Frozenset operations

Like normal sets, frozenset can also perform different operations like copy, difference, intersection, symmetric\_difference, and union.

# Frozensets

# initialize A and B

A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])

# copying a frozenset

C = A.copy() # Output: frozenset({1, 2, 3, 4})

print(C)

# union

print(A.union(B)) # Output: frozenset({1, 2, 3, 4, 5, 6})

# intersection

print(A.intersection(B)) # Output: frozenset({3, 4})

# difference

print(A.difference(B)) # Output: frozenset({1, 2})

# symmetric\_difference

print(A.symmetric\_difference(B)) # Output: frozenset({1, 2, 5, 6})

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

**Output**

frozenset({1, 2, 3, 4})

frozenset({1, 2, 3, 4, 5, 6})

frozenset({3, 4})

frozenset({1, 2})

frozenset({1, 2, 5, 6})

Similarly, other set methods like isdisjoint, issubset, and issuperset are also available.

# Frozensets

# initialize A, B and C

A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])

C = frozenset([5, 6])

# isdisjoint() method

print(A.isdisjoint(C)) # Output: True

# issubset() method

print(C.issubset(B)) # Output: True

# issuperset() method

print(B.issuperset(C)) # Output: True

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

**Output**

True

True

True

**Boolean Data Type**

**Python** **boolean** type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False. Generally, it is used to represent the truth values of the expressions. For example, 1==1 is True whereas 2<1 is False.

The boolean value can be of two types only i.e. either True or False. The output ***<class ‘bool’>*** indicates the variable is a boolean data type.

*a = True*

*print(type(a))*

*b = False*

*print(type(b))*

*# a=10  
# b=20  
# print(a>b)  
# print(type(a>b))  
#   
# a=10  
# b=20  
# print(a< b)  
# print(type(a<b))*

**Python Dictionary Data Type**

In Python, a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by ‘comma’. Dictionary holds pair of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any 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.

An effective data structure for storing data in Python is dictionaries, in which can simulate the real-life data arrangement where some specific value exists for some particular key.

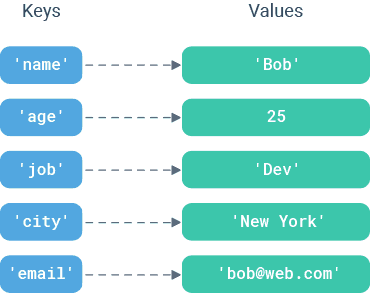
* Python Dictionary is used to store the data in a key-value pair format.
* It is the mutable data-structure.
* The elements Keys and values is employed to create the dictionary.
* Keys must consist of just one element.
* Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be of any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple. Dictionary entries are ordered as of Python version 3.7. In Python 3.6 and before, dictionaries are generally unordered.

Dictionaries are Python’s implementation of a data structure, generally known as associative arrays, hashes, or hashmaps.

You can think of a dictionary as a mapping between a set of indexes (known as keys) and a set of values. Each key maps to a value. The association of a key and a value is called a **key:value pair** or sometimes an **item**.

As an example, we’ll build a dictionary that stores employee record.



## **Create a Dictionary**

You can create a dictionary by placing a comma-separated list of key:value pairs in curly braces {}. Each key is separated from its associated value by a colon :

# Create a dictionary to store employee record

D = {'name': 'Bob',

'age': 25,

'job': 'Dev',

'city': 'New York',

'email': 'bob@web.com'}

## **The dict() Constructor**

You can convert two-value sequences into a dictionary with Python’s dict() constructor. The first item in each sequence is used as the key and the second as the value.

D1 = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})

# Create a dictionary with a list of two-item tuples

L = [('name', 'Bob'),

('age', 25),

('job', 'Dev')]

D = dict(L)

print(D)

# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}

# Create a dictionary with a tuple of two-item lists

T = (['name', 'Bob'],

['age', 25],

['job', 'Dev'])

D = dict(T)

print(D)

# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}

When the keys are simple strings, it is sometimes easier to specify key:value pairs using **keyword arguments**.

D = dict(name = 'Bob',

age = 25,

job = 'Dev')

print(D)

# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}

## **Other Ways to Create Dictionaries**

There are lots of other ways to create a dictionary.

You can use dict() function along with the zip() function, to combine separate lists of keys and values obtained dynamically at runtime.

# Create a dictionary with list of zipped keys/values

keys = ['name', 'age', 'job']

values = ['Bob', 25, 'Dev']

D = dict(zip(keys, values))

print(D)

# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}

You’ll often want to create a dictionary with default values for each key. The [fromkeys()](https://www.learnbyexample.org/python-dictionary-fromkeys-method/) method offers a way to do this.

# Initialize dictionary with default value '0' for each key

keys = ['a', 'b', 'c']

defaultValue = 0

D = dict.fromkeys(keys,defaultValue)

print(D)

# Prints {'a': 0, 'b': 0, 'c': 0}

## **Important Properties of a Dictionary**

Dictionaries are pretty straightforward, but here are a few points you should be aware of when using them.

### Keys must be unique:

A key can appear in a dictionary only once.

Even if you specify a key more than once during the creation of a dictionary, the last value for that key becomes the associated value.

D = {'name': 'Bob',

'age': 25,

'name': 'Jane'}

print(D)

# Prints {'name': 'Jane', 'age': 25}

Notice that the first occurrence of ‘name’ is replaced by the second one.

### Key must be immutable type:

You can use any object of immutable type as dictionary keys – such as numbers, strings, booleans or tuples.

D = {(2,2): 25,

True: 'a',

'name': 'Bob'}

An exception is raised when mutable object is used as a key.

# TypeError: unhashable type: 'list'

D = {[2,2]: 25,

'name': 'Bob'}

### Value can be of any type:

There are no restrictions on dictionary values. A dictionary value can be any type of object and can appear in a dictionary multiple times.

# values of different datatypes

D = {'a':[1,2,3],

'b':{1,2,3}}

# duplicate values

D = {'a':[1,2],

'b':[1,2],

'c':[1,2]}

## **Access Dictionary Items**

The order of key:value pairs is not always the same. In fact, if you write the same example on another PC, you may get a different result. In general, the order of items in a dictionary is unpredictable.

But this is not a problem because the items of a dictionary are not indexed with integer indices. Instead, you use the keys to access the corresponding values.

You can fetch a value from a dictionary by referring to its key in square brackets [].

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

print(D['name'])

# Prints Bob

If you refer to a key that is not in the dictionary, you’ll get an exception.

print(D['salary'])

print(D['salary'])

# Triggers KeyError: 'salary'

To avoid such exception, you can use the special dictionary get() method. This method returns the value for key if key is in the dictionary, else None, so that this method never raises a KeyError.

# When key is present

print(D.get('name'))

# Prints Bob

# When key is absent

print(D.get('salary'))

# Prints None

## #Accessing an element of a nested dictionary

# Creating a Dictionary

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

        'Dict2': {'Name': 'For'}}

# Accessing element using key

print(Dict['Dict1'])

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

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

## **Add or Update Dictionary Items**

Adding or updating dictionary items is easy. Just refer to the item by its key and assign a value. If the key is already present in the dictionary, its value is replaced by the new one.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

D['name'] = 'Sam'

print(D)

# Prints {'name': 'Sam', 'age': 25, 'job': 'Dev'}

If the key is new, it is added to the dictionary with its value.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

D['city'] = 'New York'

print(D)

# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev', 'city': 'New York'}

# Creating an empty Dictionary

Dict = {}

print("Empty Dictionary: ")

print(Dict)

# Adding elements one at a time

Dict[0] = 'Geeks'

Dict[2] = 'For'

Dict[3] = 1

print("\nDictionary after adding 3 elements: ")

print(Dict)

# Adding set of values

# to a single Key

Dict['Value\_set'] = 2, 3, 4

print("\nDictionary after adding 3 elements: ")

print(Dict)

# Updating existing Key's Value

Dict[2] = 'Welcome'

print("\nUpdated key value: ")

print(Dict)

# Adding Nested Key value to Dictionary

Dict[5] = {'Nested': {'1': 'Life', '2': 'Geeks'}}

print("\nAdding a Nested Key: ")

print(Dict)

## **Merge Two Dictionaries**

Use the built-in update() method to merge the keys and values of one dictionary into another. Note that this method blindly overwrites values of the same key if there’s a clash.

D1 = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

D2 = {'age': 30,

'city': 'New York',

'email': 'bob@web.com'}

D1.update(D2)

print(D1)

# Prints {'name': 'Bob', 'age': 30, 'job': 'Dev',

# 'city': 'New York', 'email': 'bob@web.com'}

## **Remove Dictionary Items**

There are several ways to remove items from a dictionary.

### Remove an Item by Key

If you know the key of the item you want, you can use pop() method. It removes the key and returns its value.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

x = D.pop('age')

print(D)

# Prints {'name': 'Bob', 'job': 'Dev'}

# get removed value

print(x)

# Prints 25

If you don’t need the removed value, use the del statement.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

del D['age']

print(D)

# Prints {'name': 'Bob', 'job': 'Dev'}

### Remove Last Inserted Item

The popitem() method removes and returns the last inserted item.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

x = D.popitem()

print(D)

# Prints {'name': 'Bob', 'age': 25}

# get removed pair

print(x)

# Prints ('job', 'Dev')

### Remove all Items

To delete all keys and values from a dictionary, use clear() method.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

D.clear()

print(D)

# Prints {}

## **Get All Keys, Values and Key:Value Pairs**

There are three dictionary methods that return all of the dictionary’s keys, values and key-value pairs: keys(), values(), and items(). These methods are useful in loops that need to step through dictionary entries one by one.

All the three methods return **iterable object**. If you want a true list from these methods, wrap them in a list() function.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

# get all keys

print(list(D.keys()))

# Prints ['name', 'age', 'job']

# get all values

print(list(D.values()))

# Prints ['Bob', 25, 'Dev']

# get all pairs

print(list(D.items()))

# Prints [('name', 'Bob'), ('age', 25), ('job', 'Dev')]

## **Iterate Through a Dictionary**

If you use a dictionary in a for loop, it traverses the keys of the dictionary by default.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

for x in D:

print(x)

# Prints name age job

To iterate over the values of a dictionary, index from key to value inside the for loop.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

for x in D:

print(D[x])

# Prints Bob 25 Dev

## **Check if a Key or Value Exists**

If you want to know whether a key exists in a dictionary, use in and not in operators with if statement.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

print('name' in D)

# Prints True

print('salary' in D)

# Prints False

To check if a certain value exists in a dictionary, you can use method values(), which returns the values as a list, and then use the in operator.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

print('Bob' in D.values())

# Prints True

print('Sam' in D.values())

# Prints False

## **Find Dictionary Length**

To find how many key:value pairs a dictionary has, use len() method.

D = {'name': 'Bob',

'age': 25,

'job': 'Dev'}

print(len(D))

# Prints 3

## **Python Dictionary Methods**

Python has a set of built-in methods that you can invoke on dictionary objects.

|  |  |
| --- | --- |
| *Python Dictionary Methods* | |
| Method | Description |
| clear() | Removes all items from the dictionary |
| copy() | Returns a shallow copy of the dictionary |
| fromkeys() | Creates a new dictionary with the specified keys and values |
| get() | Returns the value of the specified key |
| items() | Returns a list of key:value pair |
| keys() | Returns a list of all keys from dictionary |
| pop() | Removes and returns single dictionary item with specified key. |
| popitem() | Removes and returns last inserted key:value pair from the dictionary. |
| setdefault() | Returns the value of the specified key, if present. Else, inserts the key with a specified value. |
| update() | Updates the dictionary with the specified key:value pairs |
| values() | Returns a list of all values from dictionary |

# demo for all dictionary methods

dict1 = {1: "Python", 2: "Java", 3: "Ruby", 4: "Scala"}

# copy() method

dict2 = dict1.copy()

print(dict2)

# clear() method

dict1.clear()

print(dict1)

# get() method

print(dict2.get(1))

# items() method

print(dict2.items())

# keys() method

print(dict2.keys())

# pop() method

dict2.pop(4)

print(dict2)

# popitem() method

dict2.popitem()

print(dict2)

# update() method

dict2.update({3: "C++"})

print(dict2)

# values() method

print(dict2.values())

## **Built-in Dictionary Functions**

A technique that may be used on a construct to produce a value is known as a function. Additionally, it doesn't change the construct. A Python dictionary can be used with a handful of the methods that Python provides.

Python also has a set of built-in functions that you can use with dictionary objects.

|  |  |
| --- | --- |
| *Python Built-in Functions with Dictionary* | |
| Method | Description |
| all() | Returns True if all list items are true |
| any() | Returns True if any list item is true |
| len() | Returns the number of items in the list |
| sorted() | Returns a sorted list |

The built-in python dictionary methods along with the description are given below.

* **len()**

Python's len() method returns the dictionary's length. Each key-value pair lengthens the string by one.

**Code**

1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
2. len(dict)

**Output**

4

* **any()**

The any() method returns True indeed if one dictionary key does have a Boolean expression of True, much like it does for lists and tuples.

**Code**

dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}

print(any({'':'','':'',3:''}))

**Output**

**Output**

True

* **all()**

Unlike in any() method, all() only returns True if each of the dictionary's keys contain a True Boolean value.

**Code**

1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
2. print(all({1:'',2:'','':''}))  
   print(all({1:'',2:'',3:'',4:''}))

**Output**

False

True

* **sorted()**

The sorted() method returns an ordered series of the dictionary's keys, much like it does with lists as well as tuples. The initial Python dictionary is not changed by the ascending sorting.

**Code**

1. dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
2. sorted(dict)

**Output**

[ 1, 5, 7, 8]

## **Built-in Dictionary methods**

The built-in python dictionary methods along with the description and Code are given below.

* **clear()**

It is used to delete all the items of the dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # clear() method
4. dict.clear()
5. **print**(dict)

**Output**

{ }

* **copy()**

It returns a shallow copy of the dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # copy() method
4. dict\_demo = dict.copy()
5. **print**(dict\_demo)

**Output**

{1: 'Microsoft', 2: 'Google', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}

* **pop()**

eliminates the element using the defined key.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # pop() method
4. dict\_demo = dict.copy()
5. x = dict\_demo.pop(1)
6. **print**(x)

**Output**

{2: 'Google', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}

**popitem()**

removes the most recent key-value pair entered

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # popitem() method
4. dict\_demo.popitem()
5. **print**(dict\_demo)

**Output**

{1: 'Microsoft', 2: 'Google', 3: 'Facebook'}

* **keys()**

It returns all the keys of the dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # keys() method
4. **print**(dict\_demo.keys())

**Output**

dict\_keys([1, 2, 3, 4, 5])

* **items()**

It returns all the key-value pairs as a tuple.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # items() method
4. **print**(dict\_demo.items())

**Output**

dict\_items([(1, 'Microsoft'), (2, 'Google'), (3, 'Facebook'), (4, 'Amazon'), (5, 'Flipkart')])

* **get()**

It is used to get the value specified for the passed key.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # get() method
4. **print**(dict\_demo.get(3))

**Output**

Facebook

* **update()**

It updates the dictionary by adding the key-value pair of dict2 to this dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # update() method
4. dict\_demo.update({3: "TCS"})
5. **print**(dict\_demo)

**Output**

{1: 'Microsoft', 2: 'Google', 3: 'TCS'}

* **values()**

It returns all the values of the dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Microsoft", 2: "Google", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # values() method
4. **print**(dict\_demo.values())

**Output**

dict\_values(['Microsoft', 'Google', 'TCS'])

## **Conditional Statements in Python**

When we consider our real-time scenario every day, we make some decisions and based on the decisions made we will take further actions. Hence all our daily life activities depend on the decisions we make.

A similar situation arises in the programming language as well where we have to make some decisions and based on that the program will execute.

Python provides four conditional statements. we will learn about conditional statements with brief descriptions, syntax, and simple examples for each of these conditional statements.

**In Python we can achieve decision making by using the following statements:**

* if statements
* if-else statements
* elif statements
* Nested if and if-else statements
* elif ladder

In this tutorial, we will discuss all the statements in detail with some real-time examples.

#### #1) if statements

Python if statement is one of the most commonly used conditional statements in programming languages. It decides whether certain statements need to be executed or not. It checks for a given condition, if the condition is true, then the set of code present inside the ” if ” block will be executed otherwise not.

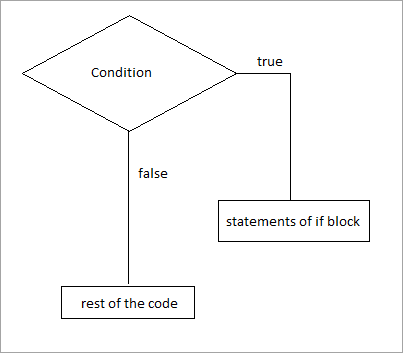
The if condition evaluates a Boolean expression and executes the block of code only when the Boolean expression becomes TRUE.

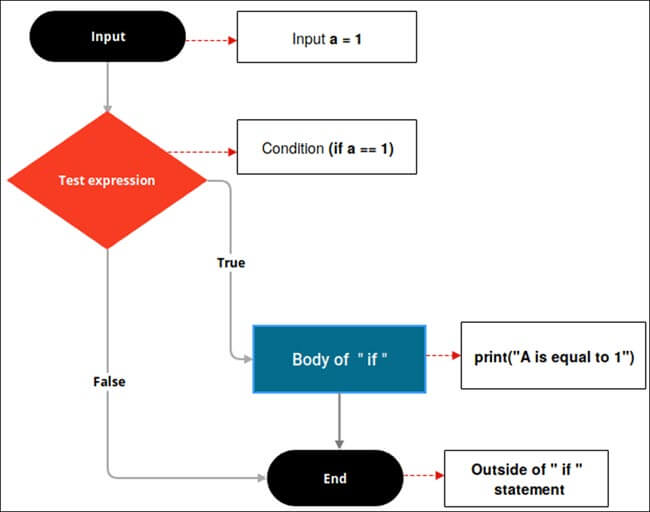
**Syntax:**

|  |
| --- |
| If ( EXPRESSION **==** TRUE ):       Block of code  **else**:       Block of code |

Here, the condition will be evaluated to a Boolean expression (true or false). If the condition is true, then the statement or program present inside the ” if ” block will be executed and if the condition is false, then the statements or program present inside the “else” block will be executed.

**Let’s see how it looks on a flow chart.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/05/if_statement_flowchart.png)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-flowchart.jpg)

If you observe the above flow-chart, first the controller will come to an if condition and evaluate the condition if it is true, then the statements will be executed, otherwise the code present outside the block will be executed.

Let’s see some examples of ” if ” statements.

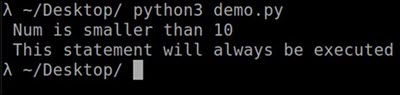
kitchen= "Apple"  
  
if (kitchen == "Apple"):  
 print("Ha Hai Apple")  
print("Thik hai")  
  
if (kitchen == "Mango"):  
 print("Ha Hai Mango")  
print("Thik hai ")  
  
if (kitchen == "Mango"):  
 print("Ha Hai Apple")

**Example: 1**

|  |
| --- |
| num = 5  if (num < 10):  print(“Num is smaller than 10”)    print(“This statement will always be executed”) |

**Output:** Num is smaller than 10.

This statement will always be executed.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example1_output-1.jpg)

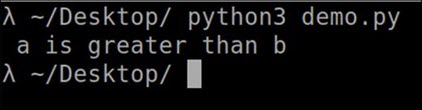
In the above example, we declared a variable called ‘Num’ with the value as 5 and the ” if ” statement is checking whether the number is lesser than 10 or not. If the condition is true then a set of statements inside the if block will be executed.

**Example: 2**

|  |
| --- |
| a **=** 7  b **=** 0  **if** (a > b):              print(“a **is** greater than b”) |

**Output:**

a is greater than b

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example2_output-1.jpg)

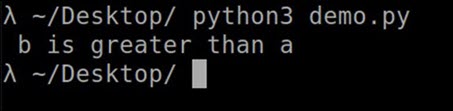
In the above example, we are checking the relationship between a and b using the greater than (>) operator in the if condition. If “a” is greater than “b” then we will get the above output.

**Example: 3**

|  |
| --- |
| a **=** 0  b **=** 7  **if** (b > a):             print(“b **is** greater than a”) |

**Output:**

b is greater than a.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example3_output-1.jpg)

**Example: 4**

|  |
| --- |
| a **=** 7  b **=** 0  **if** (a):                print(“true”) |

**Output:**

true

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example4_output-1.jpg)

If you observe, in the above example, we are not using or evaluating any condition in the “if” statement. Always remember that in any programming language, the positive integer will be treated as true value and an integer which is equal to 0 will be treated as false.

Here the value of a is 7 which is positive, hence it prints true in the console output.

**Example: 5**

# python program to illustrate If statement

i **=** 10

**if** (i > 15):

    print("10 is less than 15")

print("I am Not in if")

**Output:**

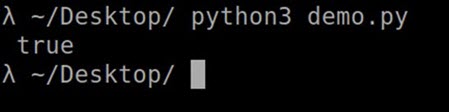
I am Not in if

**Example: 6**

|  |
| --- |
| **if** (‘Python’ **in** [‘Java', ‘Python’, ‘C#’]):       print(“true”) |

**Output:**

true

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example5_output-1.jpg)

Here, we are verifying if the element ‘Python’ is present in the given list or not. Hence it prints true because “ Python “ is present in the given list.

**Let’s take one real-life example where we will use the Python if statement.**

**For Example**: You have written an exam for a total score of 100 and if your score is above or equal to 40 then you will be considered as PASS in the exam.

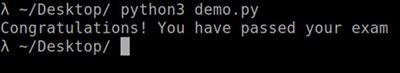
Let’s write the code for it.

**Example: 7**

|  |
| --- |
| passing\_Score **=** 40  my\_Score **=** 67  **if**(my\_Score >**=** passing\_Score):               print(“Congratulations! You have passed your exam”) |

**Output:**

Congratulations! You have passed your exam.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example6_output-1.jpg)

Remember to use the (:) operator at the end of the if statement, because whatever the code you write after the colon operator will be a part of “if block” and indentation is very important in Python.

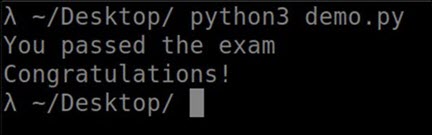
**Example: 8**

|  |
| --- |
| passing\_Score **=** 60  my\_Score **=** 67  **if**(my\_Score >**=** passing\_Score):               print(“You passed the exam”)  print(“Congratulations!”) |

**Output:**

You passed the exam

Congratulations!

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-statements-example7_output-1.jpg)

Here, print(“Congratulations!”) statement will always be executed even though the given condition is true or false.

The problem with the above code is the statement ‘print(“Congratulations!”)’ will always be executed even if the condition is evaluated to true or false. But in real-time, if you pass the exam or if you fail in the exam, then the system will say Congratulations!!!.

In order to avoid this, Python provides one conditional statement called if-else.

#### #2) if-else statements

The statement itself says if a given condition is true then execute the statements present inside the “if block” and if the condition is false then execute the “else” block.

The “else” block will execute only when the condition becomes false. It is the block where you will perform some actions when the condition is not true.

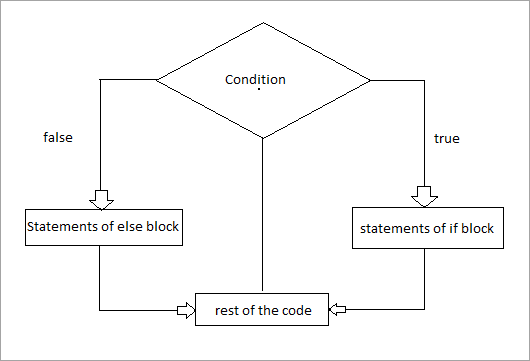
if-else statement evaluates the Boolean expression. If the condition is TRUE then, the code present in the “ if “ block will be executed otherwise the code of the “else“ block will be executed

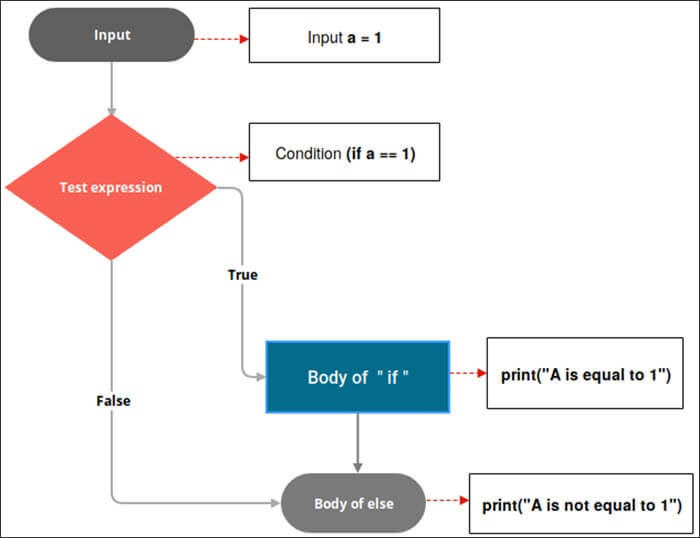
**Syntax:**

|  |
| --- |
| If (EXPRESSION **==** TRUE):          Statement (Body of the block)  **else**:          Statement (Body of the block) |

Here, the condition will be evaluated to a Boolean expression (true or false). If the condition is true then the statements or program present inside the “if” block will be executed and if the condition is false then the statements or program present inside the “else” block will be executed.

**Let’s see the flowchart of if-else**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/05/if-else_statement_flowchart.png)

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-flowchart.jpg)

If you observe the above flow chart, first the controller will come to if condition and evaluate the condition if it is true and then the statements of if block will be executed otherwise “else” block will be executed and later the rest of the code present outside “if-else” block will be executed.

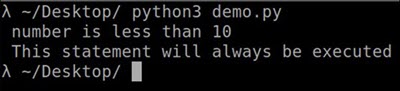
**Example: 1**

|  |
| --- |
| num **=** 5  **if**(num > 10):       print(“number **is** greater than 10”)  **else**:       print(“number **is** less than 10”)    print (“This statement will always be executed” ) |

**Output:**

number is less than 10.

This statement will always be executed.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-example1_output-1.jpg)

In the above example, we have declared a variable called ‘num’ with the value as 5 and in the “if” statement we are checking if the number is greater than 5 or not.

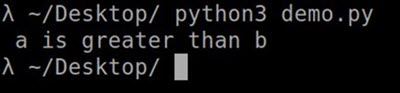
If the number is greater than 5 then, the block of code inside the “if” block will be executed and if the condition fails then the block of code present inside the “else” block will be executed.

**Example: 2**

|  |
| --- |
| a **=** 7  b **=** 0  **if** (a > b):             print(“a **is** greater than b”)  **else**:             print(“b **is** greater than a”) |

**Output:**

a is greater than b

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-example2_output-1.jpg)

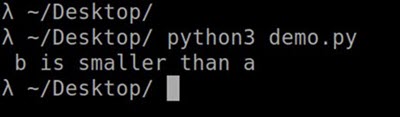
In the above code if “a” is greater than “b” then the statements present inside the “if” block will be executed and the statements present inside the “else” block will be skipped.

**Example: 3**

|  |
| --- |
| a **=** 7  b **=** 0  **if** (a < b):              print( “a **is** smaller than b” )  **else**:              print( “b **is** smaller than a” ) |

**Output:**

b is smaller than a

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-example3_output-1.jpg)

In the above code, “a” is smaller than “b”, hence statements present inside the “else” block will be executed and statements present inside the “if” block will be skipped.

Now let’s take a real-time example.

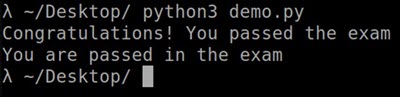
**Example: 4**

|  |
| --- |
| passing\_Score **=** 60  my\_Score **=** 67  **if**(my\_Score >**=** passing\_Score):               print(“Congratulations! You passed the exam”)               print("You are passed in the exam")  **else**:               print(“Sorry! You failed the exam, better luck next time”) |

**Output:**

Congratulations! You passed the exam

You are passed in the exam

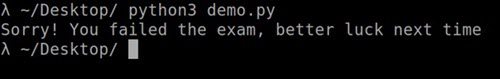
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-example4_output.jpg)

**Example: 5**

|  |
| --- |
| passing\_Score **=** 60  my\_Score **=** 47  **if**(my\_Score >**=** passing\_Score):               print(“Congratulations! You passed the exam”)               print("You are passed in the exam")  **else**:               print(“Sorry! You failed the exam, better luck next time”) |

**Output:**

Sorry! You failed the exam, better luck next time

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-example5_output.jpg)

**Example: 6**

i **=** 20

**if** (i < 15):

    print("i is smaller than 15")

    print("i'm in if Block")

**else**:

**print**("i is greater than 15")

    print("i'm in else Block")

print("i'm not in if and not in else Block")

#### #3) elif statements

In Python, we have one more conditional statement called “elif” statements. “elif” statement is used to check multiple conditions only if the given condition is false. It’s similar to an “if-else” statement and the only difference is that in “else” we will not check the condition but in “elif” we will check the condition.

“elif” statements are similar to “if-else” statements but “elif” statements evaluate multiple conditions.

**Syntax:**

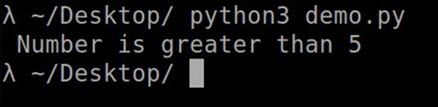
|  |
| --- |
| **if** (condition):              #Set of statement to execute if condition is true  **elif** (condition):              #Set of statements to be executed when if condition is false and elif condition is true  **else**:         #Set of statement to be executed when both if and elif conditions are false |

**Example: 1**

|  |
| --- |
| num **=** 10  **if** (num **==** 0):       print(“Number **is** Zero”)    **elif** (num > 5):         print(“Number **is** greater than 5”)    **else**:         print(“Number **is** smaller than 5”) |

**Output:**

Number is greater than 5

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/elif-statements-example1_output-1.jpg)

In the above example we have declared a variable called ‘num’ with the value as 10, and in the “if” statement we are checking the condition if the condition becomes true. Then the block of code present inside the “if” condition will be executed.

If the condition becomes false then it will check the “elif” condition if the condition becomes true, then a block of code present inside the “elif” statement will be executed.

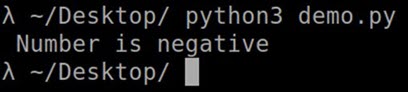
If it is false then a block of code present inside the “else” statement will be executed.

**Example: 2**

|  |
| --- |
| num **=** **-**7  **if** (num > 0):                print(“Number **is** positive”)  **elif** (num < 0):                print(“Number **is** negative”)  **else**:                print(“Number **is** Zero”) |

**Output:**

Number is negative

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/elif-statements-example2_output-1.jpg)

In the above example, first, we are assigning the value 7 to a variable called num. The controller will come to the “if” statement and evaluate the Boolean expression num > 0 but the number is not greater than zero hence if block will be skipped.

As the if condition is evaluated to false the controller will come to the “elif” statement and evaluate the Boolean expression num < 0, hence in our case number is less than zero hence ‘Number is negative’ is printed.

In case both the “if” and “elif” condition is evaluated to false then a set of statements present inside the “else” block will be executed.

#### #4) Nested if-else statements

Nested “if-else” statements mean that an “if” statement or “if-else” statement is present inside another if or if-else block. Python provides this feature as well, this in turn will help us to check multiple conditions in a given program.

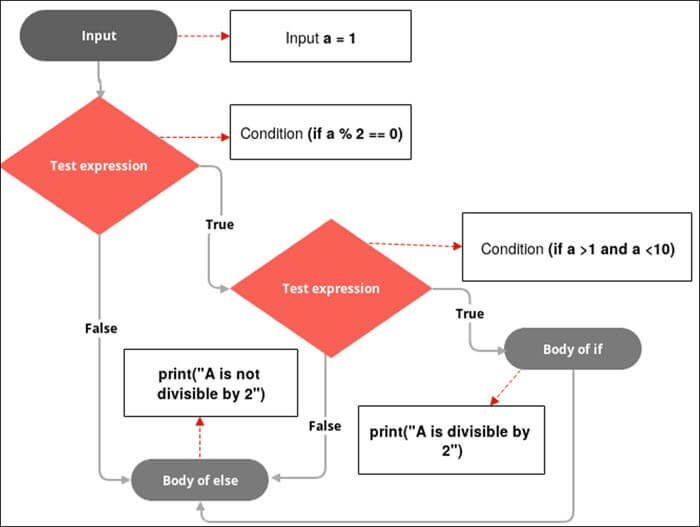
An “if” statement is present inside another “if” statement which is present inside another “if” statements and so on.

**Nested if Syntax:**

|  |
| --- |
| **if**(condition):             #Statements to execute if condition is true  **if**(condition):                      #Statements to execute if condition is true             #end of nested if  #end of if |

The above syntax clearly says that the if block will contain another if block in it and so on. If block can contain ‘n’ number of if block inside it.

**Let’s look at the nested if-else statement**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Nested-if-else-statements-flowchart.jpg)



**Example: 1**

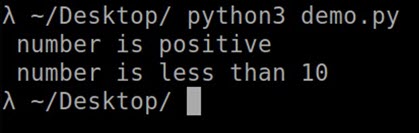


|  |
| --- |
| num **=** 5  **if**(num >0):         print(“number **is** positive”)    **if**(num<10):         print(“number **is** less than 10”) |

**Output:**

number is positive

number is less than 10

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Nested-if-else-statements-example1_output-1.jpg)

In the above example, we have declared a variable called ‘num’ with the value as 5.

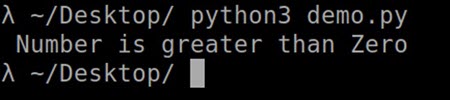
First, it will check the first “if” statement if the condition is true, then the block of code present inside the first “if” statement will be executed then it will check the second “if” statement if the first “if” statement is true and so on.

**Example: 2**

|  |
| --- |
| num **=** 7  **if** (num !**=** 0):  **if** (num > 0):                              print(“Number **is** greater than Zero”) |

**Output:**

Number is greater than Zero

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Nested-if-else-statements-example2_output-1.jpg)

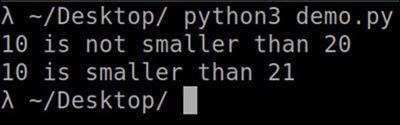
Here, the controller will check if the given number is not equal to Zero or not, if the number is not equal to zero then it enters the first if block and then in the second if block it will check if the number is greater than Zero or not, if it’s true then the control enters the nested if block and executes the statements and leaves the block and terminates the program.

**Example: 3**

|  |
| --- |
| i **=** 10  **if** (i **==** 10):  **if** (i < 20):        print (i, "is smaller than 20")  **if** (i < 21):        print (i, "is smaller than 21") |

**Output:**

10 is not smaller than 20  
10 is smaller than 21

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Nested-if-else-statements-example3_output-1.jpg)

**Example: 4**

i **=** 10

**if** (i **==** 10):

    #  First if statement

**if** (i < 15):

**print**("i is smaller than 15")

**Nested if-else Syntax:**

|  |
| --- |
| **if**(condition):             #Statements to execute if condition is true  **if**(condition):                       #Statements to execute if condition is true  **else**:                      #Statements to execute if condition is false  **else**:              #Statements to execute if condition is false |

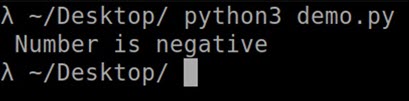
Here we have included the “if-else” block inside an if block, you can also include an “if-else” block inside “else” block.

**Example: 5**

|  |
| --- |
| num **=** **-**7  **if** (num !**=** 0):  **if** (num > 0):                        print(“Number **is** positive”)  **else**:                        print(“Number **is** negative”)  **else**:             print(“Number **is** Zero”) |

**Output:**

Number is negative

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Nested-if-else-statements-example4_output-1.jpg)

**Example: 6**

i **=** 10

**if** (i **==** 10):

    #  First if statement

**if** (i < 15):

**print**("i is smaller than 15")

    # Nested - if statement

    # Will only be executed if statement above

    # it is true

**if** (i < 12):

        print("i is smaller than 12 too")

**else**:

**print**("i is greater than 15")

**Output:**

i is smaller than 15

i is smaller than 12 too

**Example: 7**

num = 5  
if (num < 4):  
 print("Num is smaller than 10")  
 if (num<4):  
 print("Num is less than 8")  
 if (num < 5):  
 print("Num is less than 5")  
 else:  
 print("num is not less than 5")  
 else:  
 print("num is not less than 8")  
  
else:  
 print("num is not less than 10")



#### #5) elif Ladder



We have seen about the “elif” statements but what is this elif ladder? As the name itself suggests a program that contains a ladder of “elif” statements or “elif” statements are structured in the form of a ladder.



This statement is used to test multiple expressions.



**Syntax:**

|  |
| --- |
| **if** (condition):              #Set of statement to execute if condition is true  **elif** (condition):               #Set of statements to be executed when if condition is false and elif condition is true  **elif** (condition):               #Set of statements to be executed when both if and first elif condition is false and second elif condition is true  **elif** (condition):                #Set of statements to be executed when if, first elif and second elif conditions are false and third elif statement is true  **else**:         #Set of statement to be executed when all if and elif conditions are false |

**Example: 1**

i **=** 20

**if** (i **==** 10):

**print**("i is 10")

**elif** (i **==** 15):

    print("i is 15")

**elif** (i **==** 20):

**print**("i is 20")

**else**:

    print("i is not present")

**Output:**

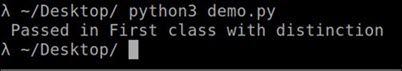
i is 20

**Example: 2**

|  |
| --- |
| my\_marks **=** 90  **if** (my\_marks < 35):                print(“Sorry!, You failed the exam”)  **elif**(my\_marks > 60 **and** my\_marks > 100):                print(“Passed **in** First **class**”)  **else**:                print(“Passed **in** First **class** with distinction”) |

**Output:**

Passed in First class with distinction

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/elif-Ladder-example1_output-1.jpg)

The above example describes the elif ladder. Firstly the control enters the “if” statement and evaluates the condition if the condition is true then the set of statements present inside the if block will be executed else it will be skipped and the controller will come to the first elif block and evaluate the condition.

A similar process will continue for all the remaining “elif” statements and in case all if and elif conditions are evaluated to false then the else block will be executed.

### Python If Statement In One Line

In Python, we can write “if” statements, “if-else” statements and “elif” statements in one line without worrying about the indentation.

We know we can write “if” statements as shown below

**Syntax:**

if (condition):

#Set of statements to execute if condition is true

In Python, it is permissible to write the above block in one line, which is similar to the above block.

**Syntax:**

if (condition): #Set of statements to execute if condition in true

There can be multiple statements as well, you just need to separate it by a semicolon (;)

**Syntax:**

if (condition): statement 1; statement 2; statement 3;…;statement n

If the condition is true, then execute statement 1, statement 2 and so on up to statement n.

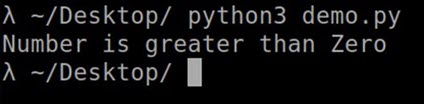
In case if the condition is false then none of the statements will be executed.

**Example: 1**

|  |
| --- |
| num **=** 7  **if** (num > 0): print(“Number **is** greater than Zero”) |

**Output:**

Number is greater than Zero

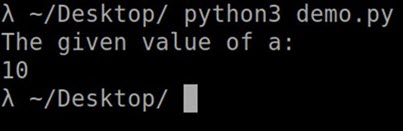
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-in-One-line-example1_output-1.jpg)

**Example: 2**

|  |
| --- |
| a **=** 10  **if** (a): print( " The given value of a: " ); print(a) |

**Output:**

The given value of a: 10

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-in-One-line-example2_output-1.jpg)

### If-else Statements In One Line

**Syntax:**

|  |
| --- |
| **if** (condition):             #Set of statement to execute if condition is true  **else**:              #Set of statement to execute if condition is false |

The above if-else block can also be written as shown below.

**Syntax:**

|  |
| --- |
| **if** (condition): #Set of statement to execute if condition is true  **else**: #Set of statement to execute if condition is false |

There can be multiple statements as well, you just need to separate it by a semicolon (;)

**Syntax:**

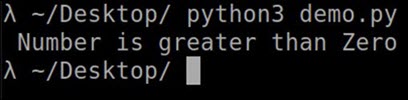
|  |
| --- |
| **if** (condition): statement 1; statement 2; statement 3;…;statement n  **else**: statement 1; statement 2; statement 3;…;statement n |

**Example: 1**

|  |
| --- |
| num **=** 7  **if** (num > 0): print(“Number **is** greater than Zero”)  **else**: print(“Number **is** smaller than Zero”) |

**Output:**

Number is smaller than Zero

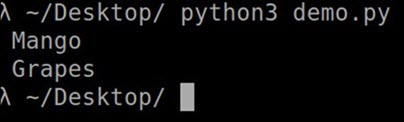
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-in-One-line-example1_output-2.jpg)

**Example: 2**

|  |
| --- |
| **if** (‘a’ **in** ‘fruits’): print(“Apple”); print(“Orange”)  **else**: print(“Mango”); print(“Grapes”) |

**Output:**

Mango  
Grapes

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/if-else-statements-in-One-line-example2_output-1.jpg)

### Elif Statements In One Line

**Syntax:**

|  |
| --- |
| **if** (condition):             #Set of statement to execute if condition is true  **elif** (condition1):               #Set of statement to execute if condition1 is true  **else**:               #Set of statement to execute if condition and condition1 is false |

The above elif block can also be written as below.

**Syntax:**

|  |
| --- |
| **if** (condition): #Set of statement to execute if condition is true  **elif** (condition1): #Set of statement to execute if condition1 is true  **else**: #Set of statement to execute if condition and condition1 is false |

There can be multiple statements as well, you just need to separate it by a semicolon (;)

**Syntax:**

|  |
| --- |
| **if** (condition): statement 1; statement 2; statement 3;…;statement n  **elif** (condition): statement 1; statement 2; statement 3;…;statement n  **else**: statement 1; statement 2; statement 3;…;statement n |

**Example: 1**

|  |
| --- |
| num **=** 7  **if** (num < 0): print("Number is smaller than Zero")  **elif** (num > 0): print("Number is greater than Zero")  **else**: print("Number is Zero") |

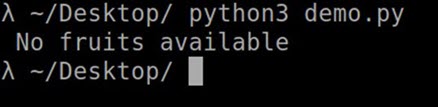
**Output:**

Number is greater than Zero

**Example: 2**

|  |
| --- |
| **if** (‘a’ **in** ‘fruits’): print(“Apple”); print(“Orange”)  **elif** (‘e’ **in** ‘fruits’): print(“Mango”); print(“Grapes”)  **else**: print(“No fruits available”) |

**Output:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/elif-Statements-in-One-Line-example2_output-1.jpg)

### Multiple Conditions In If Statements

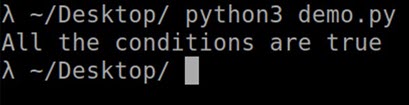
It’s not that you can only write one condition inside an “if” statement, we can also evaluate multiple conditions in an “if” statement like below.

**Example: 1**

|  |
| --- |
| num1 **=** 10  num2 **=** 20  num3 **=** 30  **if** (num1 **==** 10 **and** num2 **==** 20 **and** num3 **==** 30):               print(“All the conditions are true”) |

**Output:**

All the conditions are true

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Multiple-Conditions-in-if-Statements-example1_output-1.jpg)

Here, in the “if” statement we are checking multiple conditions using AND operator, which means if all the conditions are true only when the statements inside an if block will be executed.

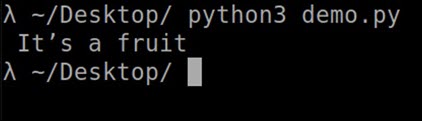
We can also specify the OR operators as well.

**Example: 2**

|  |
| --- |
| fruitName **=** “Apple”  **if** (fruitName **==** “Mango” **or** fruitName **==** “Apple” **or** fruitName **==** “Grapes”):                print(“It’s a fruit”) |

**Output:**

It’s a fruit

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Multiple-Conditions-in-if-Statements-example2_output.jpg)

Here, in an “if” statement out of three conditions, only one condition is true as that’s the rule of the OR operator. If any one condition is true then the condition will become true and the statement present inside the if block will be executed.

Let’s consider a real-time scenario to find the number of days present in a month and we know that during a leap year the number of days will change. We will see this in a programmatic way using “if, elif and else” statements.

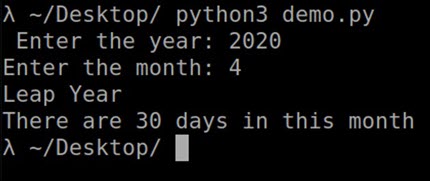
**Example: 3**

|  |
| --- |
| currentYear = int(input (" Enter the year: " ) ) month = int(input("Enter the month: " ) ) if ((currentYear % 4 ) == 0):  print("Leap Year")  if(month == 1 or month == 3 or month == 5 or month == 7 or month == 8 or month == 10 or month == 12):  print("There are 31 days in this month " )  elif ( month == 4 or month == 6 or month == 9 or month == 11 ):  print("There are 30 days in this month " )  elif ( month == 2 ):  print("There are 29 days in this month " )  else:  print("Invalid month ") elif ( ( currentYear % 4 ) != 0):  print("Non Leap Year " )  if ( month == 1 or month == 3 or month == 5 or month == 7 or month == 8 or month == 10 or month == 12 ):  print("There are 31 days in this month" )  elif ( month == 4 or month == 6 or month == 9 or month == 11 ):  print("There are 30 days in this month " )  elif ( month == 2 ):  print("There are 28 days in this month ")  else:  print("Invalid month " ) else:  print( " Invalid Year " ) |

**Output: 1**

Enter the year: 2020  
Enter the month: 4

Leap Year  
There are 30 days in this month

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2021/03/Multiple-Conditions-in-if-Statements-example3_output-1.jpg)

# Loops Statements in Python

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



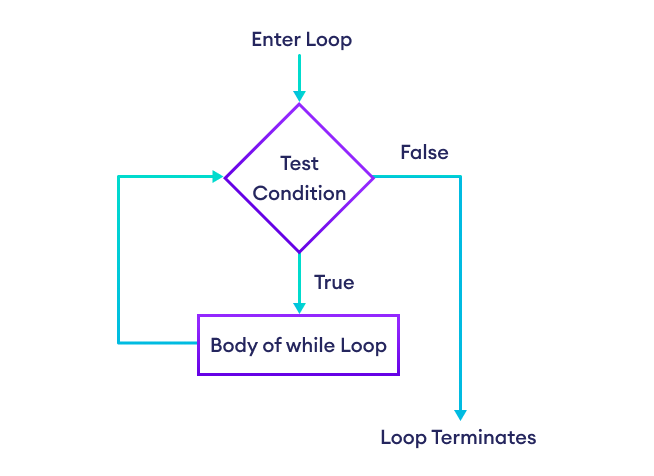
In Python, there are three different types of loops: for loop, while loop, and nested loop.

## **While Loop in Python**

In python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

The while loop is to be used in situations where the number of iterations is unknown at first. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

In Python, the while loop executes the statement or group of statements repeatedly while the given condition is True. And when the condition becomes false, the loop ends and moves to the next statement after the loop.



**Syntax:**

while expression:

statement(s)

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

1. A while loop evaluates the condition
2. If the condition evaluates to True, the code inside the while loop is executed.
3. condition is evaluated again.
4. This process continues until the condition is False.
5. When condition evaluates to False, the loop stops.

### Example of Python While Loop

kitchen= "Apple"  
while "Apple" in kitchen:  
 print("Apple")

kitchen= "Apple"  
while "Apple" in kitchen:  
 print(Apple)  
 kitchen = "Banana"



count = 0  
while (count < 3):  
 print("Credence")



count = 0  
while (count < 3):

count = count + 1  
 print("Credence")

kitchen= "Apple"  
while "Apple" in kitchen:  
 print(kitchen)  
 kitchen = "Banana"  
else:  
 print("Not Found In Kitchen")



count = 0  
while (count < 3):  
 print("Credence")

count = count + 1

else:  
 print("Not Found ")

while (count<3):  
 count = count+1  
 print("Apple")

kitchen = "Apple"  
count = 0  
  
while (count<3):  
 count = count+1  
 print("Apple")

i = 0  
while i < 6:  
 print(i)  
 i = i+1

i = 1  
while i <= 10:  
 print(i)  
 i = i+1

i = 1  
while i < 6:  
 print(i)  
 i = i+1

i = 2  
while i < 6:  
 print(i)  
 i = i+1

count = 0  
while (count < 5):  
 count += 1;  
 print("Hello Geek","Entered",count,"thtimes")

Let’s see a simple example of while loop in Python.

* Python3

|  |
| --- |
| # Python program to illustrate while loop  count **=** 0  **while** (count < 3):      count **=** count **+** 1      print("Hello Geek") |

**Output**

Hello Geek

Hello Geek

Hello Geek

### **Using else statement with While Loop in Python**

The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won’t be executed.

**Syntax of While Loop with else statement:**

while condition:

# execute these statements

else:

# execute these statements

### **Examples of While Loop with else statement**

Here is an example of while loop with else statement in Python:

* Python3

|  |
| --- |
| # Python program to illustrate  # combining else with while  count **=** 0  **while** (count < 3):      count **=** count **+** 1      print("Hello Geek")  **else**:      print("In Else Block") |

**Output**

Hello Geek

Hello Geek

Hello Geek

In Else Block

### Infinite While Loop in Python

If we want a block of code to execute infinite number of time, we can use the while loop in Python to do so.

* Python3

|  |
| --- |
| # Python program to illustrate  # Single statement while block  count **=** 0  **while** (count **==** 0):      print("Hello Geek")  age = 32  # the test condition is always True  while age > 18:  print('You can vote') |

**Note**: It is suggested **not to use** this type of loop as it is a never-ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

# program to display numbers from 1 to 5

# initialize the variable

i = 1

n = 5

# while loop from i = 1 to 5

while i <= n:

print(i)

i = i + 1

**Output**

1

2

3

4

5

Here's how the program works:

|  |  |  |
| --- | --- | --- |
| Variable | Condition: i <= n | Action |
| i = 1 n = 5 | True | 1 is printed. i is increased to **2**. |
| i = 2 n = 5 | True | 2 is printed. i is increased to **3**. |
| i = 3 n = 5 | True | 3 is printed. i is increased to **4**. |
| i = 4 n = 5 | True | 4 is printed. i is increased to **5**. |
| i = 5 n = 5 | True | 5 is printed. i is increased to **6**. |
| i = 6 n = 5 | False |  |

### Example 2: Python while Loop

# program to calculate the sum of numbers

# until the user enters zero

total = 0

number = int(input('Enter a number: '))

# add numbers until number is zero

while number != 0:

total += number # total = total + number

# take integer input again

number = int(input('Enter a number: '))

print('total =', total)

**Output**

Enter a number: 12

Enter a number: 4

Enter a number: -5

Enter a number: 0

total = 11

In the above example, the while iterates until the user enters zero. When the user enters zero, the test condition evaluates to False and the loop ends.

## Single statement while block

Just like the if block, if the while block consists of a single statement we can declare the entire loop in a single line. If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons (;).

# Python program to illustrate

# Single statement while block

count **=** 0

**while** (count < 5): count **+=** 1; print("Hello Geek")