**Language:** Language is the use of a system of communication which consists of a set of sounds or written symbols.

**Computer Language:** computer languages are systems of communication with a computer. Such languages are used to create computer code or program code, the set of instructions forming a computer program which is executed by the computer.

Computers can only execute the machine code instructions which are part of their instruction set. Because these instructions are difficult for humans to read, and writing complex programs in machine code or other low-level programming languages is a time-consuming task, most programmers write their source code in a high-level programming language. This source code is translated into machine code by a compiler or interpreter, so that the computer can execute it to perform its tasks. A compiler produces object code which is usually in machine language, but may also be in an intermediate language which is at a lower level than the source. A runtime system is often used to execute object code by linking it with commonly used libraries. Bytecode is a lower level of source which is designed for more efficient interpretation by interpreters.

**Types of programming language based on abstraction:**

Languages vary in the level of abstraction they provide from the hardware. Some programming languages provide less or no abstraction while some provide higher abstraction. Based on the levels of abstraction, they can be classified into two categories:

1. Low-level language
2. High-level language

**Low-level language:**

The low-level language is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions.

It’s fast in execution but difficult for developer to understand, debug and write program.

The languages that come under this category are the Machine level language and Assembly language.

Machine-level language:

The machine-level language is a language that consists of a set of instructions that are in the binary form 0 or 1. As we know that computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1, so the instructions given to the computer can be only in binary codes. Creating a program in a machine-level language is a very difficult task as it is not easy for the programmers to write the program in machine instructions. It is error-prone as it is not easy to understand, and its maintenance is also very high. A machine-level language is not portable as each computer has its machine instructions, so if we write a program in one computer will no longer be valid in another computer.

Assembly Language:

The assembly language contains some human-readable commands such as mov, add, sub, etc. The problems which we were facing in machine-level language are reduced to some extent by using an extended form of machine-level language known as assembly language. Since assembly language instructions are written in English words like mov, add, sub, so it is easier to write and understand.

As we know that computers can only understand the machine-level instructions, so we require a translator that converts the assembly code into machine code. The translator used for translating the code is known as an assembler.

The assembly language code is not portable because the data is stored in computer registers, and the computer has to know the different sets of registers.

The assembly code is not faster than machine code because the assembly language comes above the machine language in the hierarchy, so it means that assembly language has some abstraction from the hardware while machine language has zero abstraction.

**High-Level Language:**

The high-level language is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine-level languages.

It’s slow in execution in compare with low level language but easy for developer to understand, debug and write program.

When writing a program in a high-level language, then the whole attention needs to be paid to the logic of the problem.

A compiler is required to translate a high-level language into a low-level language.

**Advantages of a high-level language:**

1. The high-level language is easy to read, write, and maintain as it is written in English like words.
2. The high-level languages are designed to overcome the limitation of low-level language, i.e., portability. The high-level language is portable; i.e., these languages are machine-independent.

The following are the differences between low-level language and high-level language:

|  |  |
| --- | --- |
| **Low-level language** | **High-level language** |
| It is a machine-friendly language, i.e., the computer understands the machine language, which is represented in 0 or 1. | It is a user-friendly language as this language is written in simple English words, which can be easily understood by humans. |
| It executes at a faster pace. | The high-level language takes more time to execute. |
| It requires the assembler to convert the assembly code into machine code. | It requires the compiler to convert the high-level language instructions into machine code. |
| The machine code cannot run on all machines, so it is not a portable language. | The high-level code can run all the platforms, so it is a portable language. |
| It is memory efficient. | It is less memory efficient. |
| Debugging and maintenance are not easier in a low-level language. | Debugging and maintenance are easier in a high-level language. |

**Types of Programming Languages based on features and coding style:**

The different types of programming languages are discussed below.

**Procedural Programming Language:**

The procedural programming language is used to execute a sequence of statements which lead to a result. Typically, this type of programming language uses multiple variables, heavy loops and other elements, which separates them from functional programming languages. Functions of procedural language may control variables, other than function’s value returns. For example, printing out information.

**Functional Programming Language:**

Functional programming language typically uses stored data, frequently avoiding loops in favor of recursive functions. The functional programing’s primary focus is on the return values of functions, and side effects and different suggests that storing state are powerfully discouraged. For example, in an exceedingly pure useful language, if a function is termed, it’s expected that the function not modify or perform any o/p. It may, however, build algorithmic calls and alter the parameters of these calls. Functional languages are usually easier and build it easier to figure on abstract issues, however, they’ll even be “further from the machine” therein their programming model makes it difficult to know precisely, but the code is decoded into machine language (which are often problematic for system programming). Example Lisp, Python, Erlang, Haskell, Clojure, etc

**Object-oriented Programming Language:**

This programming language views the world as a group of objects that have internal data and external accessing parts of that data. The aim this programming language is to think about the fault by separating it into a collection of objects that offer services which can be used to solve a specific problem. One of the main principle of object oriented programming language is encapsulation that everything an object will need must be inside of the object. This language also emphasizes reusability through inheritance and the capacity to spread current implementations without having to change a great deal of code by using polymorphism.

**Scripting Programming Language:**

These programming languages are often procedural and may comprise object-oriented language elements, but they fall into their own category as they are normally not full-fledged programming languages with support for development of large systems. For example, they may not have compile-time type checking. Usually, these languages require tiny syntax to get started.

**Difference between Procedural Programming and Object Oriented Programming:**

|  |  |
| --- | --- |
| **PROCEDURAL ORIENTED PROGRAMMING** | **OBJECT ORIENTED PROGRAMMING** |
| In procedural programming, program is divided into small parts called functions. | In object oriented programming, program is divided into small parts called objects. |
| Procedural programming follows top down approach. | Object oriented programming follows bottom up approach. |
| There is no access specifier in procedural programming. | Object oriented programming have access specifiers like private, public, protected etc. |
| Procedural programming does not have any proper way for hiding data so it is less secure. | Object oriented programming provides data hiding so it is more secure. |
| In procedural programming, overloading is not possible. | Overloading is possible in object oriented programming. |
| In procedural programming, function is more important than data. | In object oriented programming, data is more important than function. |
| Examples: C, FORTRAN, Pascal etc. | Examples: C++, Java, Python, C# etc. |

**Object-oriented programming concept:**

Following are object-oriented concepts:

1. **Class:**

Classes are user defined data type which is combination of data as well as functions. It’s just a template, no any memory allocated after declaration. Memory allocated for its variable called object. You can define a blueprint for an object.

1. **Object:**

Run time entity of class is called as object.

1. **Data Abstraction:**

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

1. **Data Encapsulation:**

The wrapping up of data and functions into a single unit (class) is called encapsulation.

1. **Inheritance:**

One of the most useful aspects of object-oriented programming is code reusability. Inheritance is the process by which objects of one class acquires the properties of objects of another class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

1. **Polymorphism:**

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

1. **Dynamic Binding:**

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphism reference depends on the dynamic type of that reference.

**Python Features**

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

1) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type print("Hello World"). It will take only one line to execute, while Java or C takes multiple lines.

2) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

3) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

4) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

5) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

6) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

7) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

8) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

9) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

10) Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

11) Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to x, then we don't need to write int x = 15. Just write x = 15.

**Python History and Versions**

* Python laid its foundation in the late 1980s.
* The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.
* In February 1991, Guido Van Rossum published the code (labeled version 0.9.0) to alt.sources.
* In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
* Python 2.0 added new features such as list comprehensions, garbage collection systems.
* On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
* ABC programming language is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.
* The following programming languages influence Python:

ABC language.

Modula-3

**Why the Name Python?**

There is a fact behind choosing the name Python. Guido van Rossum was reading the script of a popular BBC comedy series "Monty Python's Flying Circus". It was late on-air 1970s.

Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the "Monty Python's Flying Circus" for their newly created programming language.

The comedy series was creative and well random. It talks about everything. Thus it is slow and unpredictable, which made it very interesting.

**Python Libraries and Frameworks**

Python consists of vast libraries and various frameworks. After getting familiar with Python's basic concepts, the next step is to explore the Python libraries. Libraries are essential to work with the domain specific projects.

There are many libraries in Python. Below mentioned a few of them.

* TensorFlow - It is an artificial intelligence library which allows us to create large scale AI based projects.
* Django - It is an open source framework that allows us to develop web applications. It is easy, flexible, and simple to manage.
* Flask - It is also an open source web framework. It is used to develop lightweight web applications.
* Pandas - It is a Python library which is used to perform scientific computations.
* Keras - It is an open source library, which is used to work around the neural network.

**Usage of Python:**

Python is a general purpose, open source, high-level programming language and also provides number of libraries and frameworks. Python has gained popularity because of its simplicity, easy syntax and user-friendly environment. The usage of Python as follows.

* Desktop Applications
* Web Applications
* Data Science
* Artificial Intelligence
* Machine Learning
* Scientific Computing
* Robotics
* Gaming
* Mobile Apps
* Data Analysis and Preprocessing

**Python Programming**

**Python Comments**

There are 3 ways of creating comments in Python.

# This is a comment

##single line comment

"""This is a

multiline

comment."""

'''This is also a

multiline

comment.'''

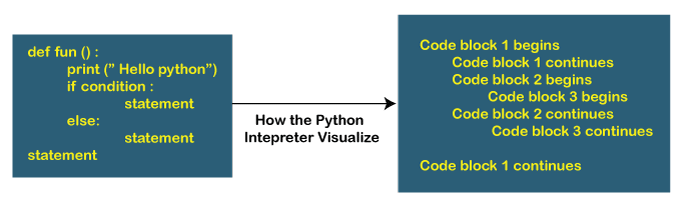
**Indentation and Comment in Python**

Indentation is the most significant concept of the Python programming language. Improper use of indentation will end up "IndentationError" in our code.

Indentation is nothing but adding whitespaces before the statement when it is needed. Without indentation Python doesn't know which statement to be executed to next. Indentation also defines which statements belong to which block. If there is no indentation or improper indentation, it will display "IndentationError" and interrupt our code.

Python indentation defines the particular group of statements belongs to the particular block. The programming languages such as C, C++, java use the curly braces {} to define code blocks.

In Python, statements that are the same level to the right belong to the same block. We can use four whitespaces to define indentation.



Example -

list1 = [1, 2, 3, 4, 5]

for i in list1:

print(i)

if i==4:

break

print("End of for loop")

Output:

1

2

3

4

End of for loop

**Multi-Line Statements**

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

total = item\_one + \

item\_two + \

item\_three

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

days = ['Monday', 'Tuesday','Wednesday', 'Thursday', 'Friday']

**Multiple Statements on a Single Line**

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

Num1=5;num2=6; num3=num1+num2;

**Tokens of Python:**

Smallest individual unit of python program is called token.

1. Keywords
2. Identifiers
3. Literals/Constants
4. Operators

**Keyword:** Reserve words provided by python and having fixed meaning. We can’t use these words for other purposes. All the Python keywords contain lowercase letters only.

Following are example of python keywords:

|  |  |  |
| --- | --- | --- |
| And | exec | not |
| Assert | finally | or |
| Break | for | pass |
| Class | from | print |
| continue | global | raise |
| Def | if | return |
| Del | import | try |
| Elif | in | while |
| Else | is | with |
| except | lambda | Yield |

**Identifiers:**

Python identifiers refer to a name used to identify a variable, function, module, class, module or other objects.

There are few rules to follow while naming the Python Variable:

* A variable name must start with either an English letter or underscore (\_).
* A variable name cannot start with the number.
* Special characters are not allowed in the variable name.
* The variable's name is case sensitive.
* Keywords can’t be used as variable name.

Starting an identifier with a single leading underscore indicates that the identifier is private.

Starting an identifier with two leading underscores indicates a strongly private identifier.

If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

**Python Literals/constant:**

Python Literals can be defined as data that is given in a variable or constant.

Python supports the following literals:

**1. String literals:**

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Ex: “abc” , “12”, “2a”, ‘5’, ‘abc’, ‘’’abc’’’, “””abc”””

**Types of Strings**:

There are two types of Strings supported in Python:

a) **Single-line String**- Strings that are terminated within a single-line are known as Single line Strings.

Example: text1='hello'

b) **Multi-line String** - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

1) Adding black slash at the end of each line.

Example:

text1='hello\

world'

print(text1)

Output:

'helloworld'

2) Using triple quotation marks:-

Example:

str2='''welcome

to

Pune'''

Print(str2)

Output:

welcome

to

Pune

**2. Numeric literals:**

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

* **Int(signed integers):** Numbers( can be both positive and negative) with no fractional part.eg: 100
* **float(floating point)**: Real numbers with both integer and fractional part eg: -26.2
* **Complex(complex)**: In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: 3.14j

**3. Boolean literals:**

A Boolean literal can have any of the two values: True(1) or False(0).

Example - Boolean Literals

x = (1 == True)

y = (2 == False)

z = (3 == 1)

a = True + 10

b = False + 10

print("x is", x)

print("y is", y)

print("z is", z)

print("a:", a)

print("b:", b)

Output:

x is True

y is False

z is False

a: 11

b: 10

**4. Special literals:**

Python contains one special literal i.e., None.

None is used to specify to that field that is not created. It is also used for the end of lists in Python.

Example - Special Literals

val1=10

val2=None

print(val1)

print(val2)

Output:

10

None

**Python Operators:**

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Python provides a variety of operators, which are described as follows.

* Arithmetic operators
* Comparison operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

**Arithmetic Operators:**

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), \*(multiplication), /(divide), %(reminder), //(floor division), and exponent (\*\*) operators.

+ (Addition) It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30

- (Subtraction) It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative.

For example, if a = 20, b = 10 => a - b = 10

/ (divide) It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0

\* (Multiplication) It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a \* b = 200

% (reminder) It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0

\*\* (Exponent) It is an exponent operator represented as it calculates the first operand power to the second operand.

// (Floor division) It gives the floor value of the quotient produced by dividing the two operands.

**Comparison operator**

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly.

== If the value of two operands is equal, then the condition becomes true.

!= If the value of two operands is not equal, then the condition becomes true.

<= If the first operand is less than or equal to the second operand, then the condition becomes true.

>= If the first operand is greater than or equal to the second operand, then the condition becomes true.

> If the first operand is greater than the second operand, then the condition becomes true.

< If the first operand is less than the second operand, then the condition becomes true.

**Assignment Operators:**

The assignment operators are used to assign the value of the right expression to the left operand.

= It assigns the value of the right expression to the left operand.

+= It increases the value of the left operand by the value of the right operand and assigns the modified value

back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a=30.

-= It decreases the value of the left operand by the value of the right operand and assigns the modified value

back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.

\*= It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a\* = b will be equal to a = a\* b and therefore, a = 200.

%= It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.

\*\*= a\*\*=b will be equal to a=a\*\*b, for example, if a = 4, b =2, a\*\*=b will assign 4\*\*2 = 16 to a.

//= A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

**Logical Operators:**

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

and If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.

or If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.

not If an expression a is true, then not (a) will be false and vice versa.

**Bitwise Operators**

The bitwise operators perform bit by bit operation on the values of the two operands.

& (binary and) If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise,

0 is copied.

| (binary or) The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.

^ (binary xor) The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0.

~ (negation) It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1

and vice versa.

<< (left shift) The left operand value is moved left by the number of bits present in the right operand.

>> (right shift) The left operand is moved right by the number of bits present in the right operand.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **0** | **0** | **1** |

**Membership Operators**

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

in It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).

not in It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

x = 'Hello world'

y = {1:'a',2:'b'}

# Output: True

print('H' in x)

# Output: True

print('hello' not in x)

# Output: True

print(1 in y)

# Output: False

print('a' in y)

**Identity Operators**

The identity operators are used to decide whether an element certain class or type.

is It is evaluated to be true if the reference present at both sides point to the same object.

is not It is evaluated to be true if the reference present at both sides do not point to the same object.

x1 = 5

y1 = 6

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

# Output: False

print(x1 is not y1)

# Output: True

print(x2 is y2)

# Output: False

print(x3 is y3)

a=6+5/4

**Operator Precedence**

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in Python is given below.

|  |  |
| --- | --- |
| Operator | Description |
| \*\* | The exponent operator is given priority over all the others used in the expression. |
| ~ + - | The negation, unary plus, and minus. |
| \* / % // | The multiplication, divide, modules, reminder, and floor division. |
| + - | Binary plus, and minus |
| >> << | Left shift. and right shift |
| & | Binary and. |
| ^ | | Binary xor, and or |
| <= < > >= | Comparison operators (less than, less than equal to, greater than, greater then equal to). |
| <> == != | Equality operators. |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

**Python Variables**

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.

**Identifier Naming**

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier 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).
* Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, \*).
* Identifier name must not be similar to any keyword defined in the language.
* Identifier names are case sensitive; for example, my name, 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.

**Declaring Variable and Assigning Values**

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

**Object References**

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class. Consider the following example.

print("John")

Output:

John

In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type() function**.

type("John")

Output:

<class 'str'>

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

Let's understand the following example

a = 50



In the above image, the variable a refers to an integer object.

Suppose we assign the integer value 50 to a new variable b.

a = 50

b = a

Python Variables

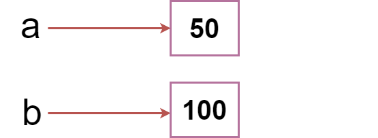
The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

a = 50

b=50

b=100



Python manages memory efficiently if we assign the same variable to two different values.

**Object Identity**

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in **id() function**, is used to identify the object identifier.

Example:

a = 50

b = a

print(id(a))

print(id(b))

# Reassigned variable a

a = 500

print(id(a))

Output:

140734982691168

140734982691168

2822056960944

We assigned the b = a, a and b both point to the same object. When we checked by the **id() function** it returned the same number. We reassign a to 500; then it referred to the new object identifier.

The variable name should be descriptive to make code more readable.

**The multi-word keywords can be created by the following method.**

**Camel Case** - In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.

**Pascal Case** - It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.

**Snake Case** - In the snake case, Words are separated by the underscore. For example - name\_of\_student, etc.

**Multiple Assignment**

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables.

1. **Assigning single value to multiple variables**

Example:

X=50

Y=50

Z=50

x=y=z=50

print(x)

print(y)

print(z)

Output:

50

50

50

2. **Assigning multiple values to multiple variables:**

Example:

A=5

B=10

C=15

a,b,c=5,10,15

print(a)

print (b)

print (c)

Output:

5

10

15

The values will be assigned in the order in which variables appear.

**Delete a variable**

We can delete the variable using the **del keyword**. The syntax is given below.

Syntax:

del <variable\_name>

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error "variable x is not defined". The variable x will no longer use in future.

Example:

x = 6

print(x)

del x

print(x)

Output:

6

Traceback (most recent call last):

File "C:/test.py", line 6, in

print(x)

NameError: name 'x' is not defined

**Maximum Possible Value of an Integer in Python**

Unlike the other programming languages, Python doesn't have long int data types. It treats all integer values as an int data type. Here, the question arises. What is the maximum possible value can hold by the variable in Python?

There is no limitation number by bits and we can expand to the limit of our memory. Python doesn't have any special data type to store larger numbers.

Example -

a = 10000000000000000000000000000000000000000000

a = a + 1

print(type(a))

print (a)

Output:

<class 'int'>

10000000000000000000000000000000000000000001

**Print Single and Multiple Variables in Python**

We can print multiple variables within the single print statement. Below are the example of single and multiple printing values.

Example - 1 (Printing Single Variable)

a = 5

print(a)

print((a))

Output:

5

5

Example - 2 (Printing Multiple Variables)

a = 5

b = 6

print(a,b)

Print(1, 2, 3, 4, 5, 6, 7, 8)

Output:

5 6

1 2 3 4 5 6 7 8

**Python Data Types**

Datatype tells about size and type of data. Based on the data type of a variable, the operating system allocates memory of specific size and decides what can be stored in the reserved memory.

**Python Data Types**

Variables can hold values, and every value has a data-type. **Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type**.

a = 5

The variable a holds integer value five and we did not define its type. Python interpreter will automatically interpret variables a as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type() function**, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

a=10

b="Hi Python"

c = 10.5

print(type(a))

print(type(b))

print(type(c))

Output:

<class 'int'>

<class 'str'>

<class 'float'>

Example:

c = 5 + 3j

print(c + 3)

print(isinstance(c, complex))

Output:

(8+3j)

True

**Standard data types**

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

Text Type: str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

Binary Types: bytes, bytearray, memoryview

**Setting the Data Type**

In Python, the data type is set when you assign a value to a variable:

|  |  |
| --- | --- |
| Example | Data Type |
| x = "Hello World" | str |
| x = 20 | int |
| x = 20.5 | float |
| x = 1j | complex |
| x = ["apple", 5, "cherry"] | list |
| x = ("apple", 25, "cherry") | tuple |
| x = range(6) | range |
| x = {"name" : "John", "age" : 36} | dict |
| x = {"apple", "banana", "cherry"} | set |
| x = frozenset({"apple", "banana", "cherry"}) | frozenset |
| x = True | bool |
| x = b"Hello" | bytes |
| x = bytearray(5) | bytearray |
| x = memoryview(bytes(5)) | memoryview |

**Setting the Specific Data Type to variable:**

If you want to specify the data type to a variable, you can use the following constructor functions:

|  |  |
| --- | --- |
| **Example** | **Data Type** |
| x = str("Hello World") | str |
| x = int(20) | int |
| x = float(20.5) | float |
| x = complex(1j) | complex |
| x = list(("apple", "banana", "cherry")) | list |
| x = tuple(("apple", "banana", "cherry")) | tuple |
| x = range(6) | range |
| x = dict(name="John", age=36) | dict |
| x = set(("apple", "banana", "cherry")) | set |
| x = frozenset(("apple", "banana", "cherry")) | frozenset |
| x = bool(5) | bool |
| x = bytes(5) | bytes |
| x = bytearray(5) | bytearray |
| x = memoryview(bytes(5)) | memoryview |

**Type Conversion**

We can convert one type of number into another. This is also known as coercion.

**Implicit conversion**

Operations like addition, subtraction coerce integer to float implicitly (automatically), if one of the operands is float.

>>> 1 + 2.0

3.0

We can see above that 1 (integer) is coerced into 1.0 (float) for addition and the result is also a floating point number.

**Explicit Conversion:**

We can also use built-in functions like int(), float() etc.. to convert between types explicitly.

int(4.5)

int(“2”)

a=”2”

b=”34”

c=a+b

234

c=int(a)+int(b)

36

**To convert between types, you simply use the type name as a function.**

A=str(2)

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| Sr.No. | Function & Description |
| 1 | int(x [,base])  Converts x to an integer. base specifies the base if x is a string. |
| 2 | float(x)  Converts x to a floating-point number. |
| 3 | complex(real [,imag])  Creates a complex number. |
| 4 | str(x)  Converts object x to a string representation. |
| 6 | repr(x)  Converts object x to an expression string. |
| 7 | eval(str)  Evaluates a string and returns an object. |
| 8 | tuple(s)  Converts s to a tuple. |
| 9 | list(s)  Converts s to a list. |
| 10 | set(s)  Converts s to a set. |
| 11 | dict(d)  Creates a dictionary. d must be a sequence of (key,value) tuples. |
| 12 | frozenset(s)  Converts s to a frozen set. |
| 13 | chr(x)  Converts an integer to a character. |
| 14 | unichr(x)  Converts an integer to a Unicode character. |
| 15 | ord(x)  Converts a single character to its integer value. |
| 16 | hex(x)  Converts an integer to a hexadecimal string. |
| 17 | oct(x)  Converts an integer to an octal string. |

**Numbers or Numeric Type:**

Python supports three types of numeric data.

int - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to int

float - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.

complex - A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

**String**

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" python" returns "hello python".

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

The following example illustrates the string in Python.

Example - 1

str = "string using double quotes"

print(str)

s = '''A multiline

string'''

print(s)

Output:

string using double quotes

A multiline

string

Example - 2

str1 = 'hello Python'

str2 = ' how are you'

print (str1[0:2])

print (str1[4])

print (str1\*2)

print (str1 + str2)

Output:

he

o

hello Pythonhello Python

hello Pythonhow are you

**Boolean**

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class **bool**. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.

Example.

print(type(True))

print(type(False))

print(false)

Output:

<class 'bool'>

<class 'bool'>

NameError: name 'false' is not defined

Example - 2

x = (1 == True)

y = (2 == False)

z = (3 == 1)

a = True + 10

b = False + 10

print("x is", x)

print("y is", y)

print("z is", z)

print("a:", a)

print("b:", b)

Output:

x is True

y is False

z is False

a: 11

b: 10

**Python print() Function:**

The print() function prints the specified message to the screen, or other standard output device.

The message can be a string, or any other object, the object will be converted into a string before written to the screen.

**Syntax**

print(object(s), sep=separator, end=end, file=file, flush=flush)

**Parameter Values**

Parameter Description

object(s) Any object, and as many as you like. Will be converted to string before printed

sep='separator' Optional. Specify how to separate the objects, if there is more than one. Default is ' '

end='end' Optional. Specify what to print at the end. Default is '\n' (line feed)

file Optional. An object with a write method. Default is sys.stdout

flush Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default

is False

Example 1

print("Python is programming language.")

x = 7

print("x =", x)

y = x

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

Output:

Python is programming language.

x = 7

x = 7 = y

Explanation:

In the above code, only objects parameter is passed to print() function (in all three print statements).

The end parameter '\n' (newline character) is used to display output in the next line, and it is by default. As we can see, each print statement displays output in the new line.

If the file is saved as sys.stdout, then, the output is printed on the screen.

Here the value of flush is False, so the stream is not forcibly flushed.

Example 2

x = 7

print("x =", x, sep='00000', end='\n\n')

print("x =", x, sep='0', end='')

Output:

a =000007

a =07

**Print value of variable inside the string:**

**Syntax**

print(f”string1{var1}string2{var2})

Example:

num1=5

num2=10

print(f"Value of num1={num1} and value of num2={num2}");

Output:

Value of num1=5 and value of num2=10

**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. It throws an error EOFError if EOF is read.

Syntax: 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.

Example:

val = input("Enter a value: ")

print("You entered:",val)

Output:

Enter a value: 45

You entered: 45

Example: The input() method returns string value. So, if we want to perform arithmetic operations, we need to cast the value first.

val = input("Enter an integer: ")

val = int(val)

sqr = (val\*val)

print("Square of the value:",sqr)

Output:

Enter an integer: 12

Square of the value: 144

Example:

print('Enter your name:')

x = input()

print('Hello, ' + x)

Output:

Enter your name:Test

Hello, Test

**Decision Making and Branching**

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

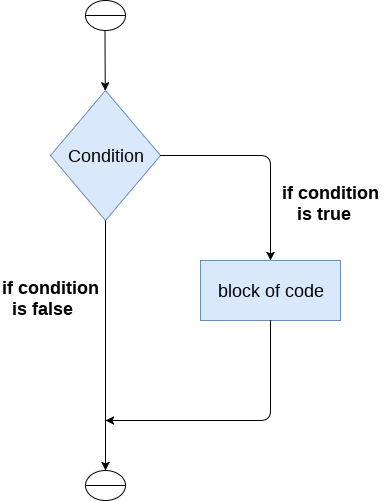
In programming the order of execution of instructions may have to be changed depending on certain conditions. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain instructions (Branch) accordingly.

Python handles decision-making and branching by supporting the following statements, if statement, if..else, elif and nested if .

**Note:** **Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value**.

**The if statement**

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

if expression:

statement

Example:

num = int(input("enter the number?"))

if num%2 == 0:

print("Number is even")

Output:

enter the number?10

Number is even

Example: Program to print the largest of the three numbers.

a = int(input("Enter a? "));

b = int(input("Enter b? "));

c = int(input("Enter c? "));

if a>b and a>c:

print("a is largest");

if b>a and b>c:

print("b is largest");

if c>a and c>b:

print("c is largest");

Output:

Enter a? 100

Enter b? 120

Enter c? 130

c is largest

**Example**

If statement, without indentation (will raise an error):

a = 33

b = 200

if b > a:

print("b is greater than a") # you will get an error

**The pass Statement**

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

Example

a = 33

b = 200

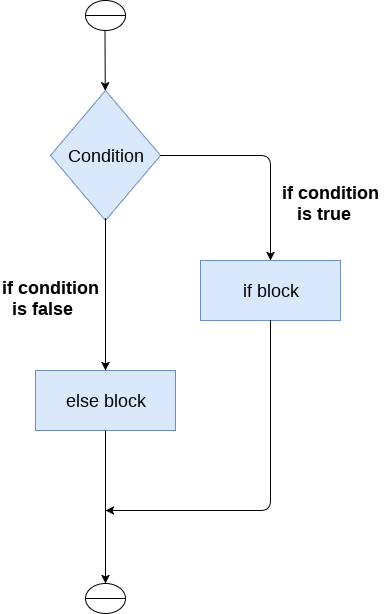
if b > a:

pass

**The if-else statement**

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

if condition:

#block of statements

else:

#another block of statements (else-block)

Example 1 : Program to check whether a person is eligible to vote or not.

age = int (input("Enter your age? "))

if age>=18:

print("You are eligible to vote !!");

else:

print("Sorry! you have to wait !!");

Output:

Enter your age? 90

You are eligible to vote !!

Example 2: Program to check whether a number is even or not.

num = int(input("enter the number?"))

if num%2 == 0:

print("Number is even...")

else:

print("Number is odd...")

Output:

enter the number?10

Number is even

**Single Statement with if or else:**

If the if clause consists only of a single line, it may go on the same line as the header statement.

Example:

var = 100

if ( var == 100 ) : print("Value of expression is 100")

print("Outside of if")

Output

Value of expression is 100

Outside of if

**Short Hand If ... Else**

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

a = 2

b = 330

print("A") if a > b else print("B")

**You can also have multiple else statements on the same line:**

Example

One line if else statement, with 3 conditions:

a = 330

b = 330

print("A") if a > b else print("=") if a == b else print("B")

**The elif statement**

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement :

if expression 1:

# block of statements

elif expression 2:

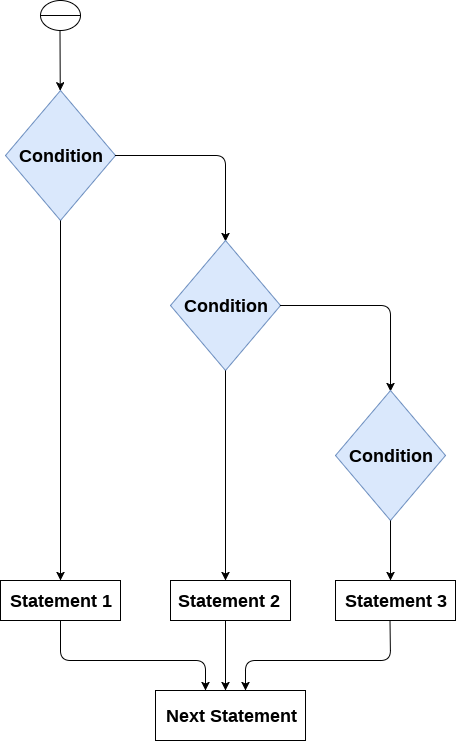
# block of statements

elif expression 3:

# block of statements

else:

# block of statements



Example:

number = int(input("Enter the number?"))

if number==10:

print("number is equals to 10")

elif number==50:

print("number is equal to 50");

elif number==100:

print("number is equal to 100");

else:

print("number is not equal to 10, 50 or 100");

Output:

Enter the number?15

number is not equal to 10, 50 or 100

**Nested if Statement**

Nested if statements enable us to use if..else statement inside an outer if statement.

The syntax of the nested if...elif...else construct may be −

if expression1:

statement(s)

if expression2:

statement(s)

elif expression3:

statement(s)

elif expression4:

statement(s)

else:

statement(s)

else:

statement(s)

Example:

num1=int(input("Enter a No:"))

num2=int(input("Enter a No:"))

num3=int(input("Enter a No:"))

if num1>num2:

if num1>num3:

print(num1," is greatest number")

else:

print(num3," is greatest number")

else:

if num2>num3:

print(num2," is greatest number")

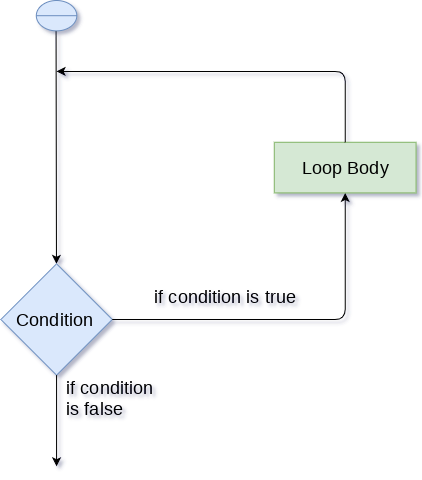
else:

print(num3," is greatest number")

**Decision Making and Looping**

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 odd numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

**There are the following loop statements in Python:**

|  |  |
| --- | --- |
| Loop Statement | Description |
| for loop | The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance. |
| while loop | The while loop is to be used in the scenario where we don't know the number of iterations in advance. 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. |
| do-while loop | The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs). **do-while loop is not available in Python**. |

**Python for loop**

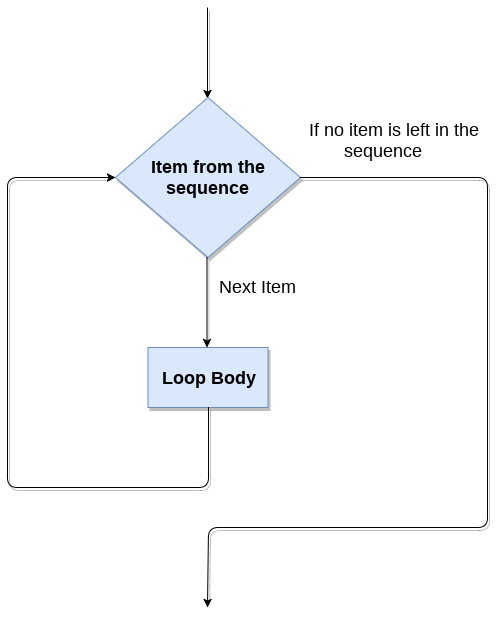
The for loop in Python is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop :

for iterating\_var in sequence:

statement(s)

The for loop flowchart



**For loop Using Sequence**

Example-1: Iterating string using for loop

str = "Python"

for i in str:

print(i)

Output:

P

y

t

h

o

n

Example- 2: Program to print the table of the given number .

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

n = 5

for i in list:

c = n\*i

print(c, end=” “)

Output:

5 10 15 20 25 30 35 40 45 50

Example-4: Program to print the sum of the given list.

list = [10,30,23,43,65,12]

sum = 0

for i in list:

sum = sum+i

print("The sum is:",sum)

Output:

The sum is: 183

**For loop Using range() function**

**The range() function**

The range() function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9. The syntax of the range() function is given below.

Syntax:

range(start, stop, step\_size)

range(10)

0 1 2 3 4 5 6 7 8 9

* The start represents the beginning of the iteration.
* The stop represents that the loop will iterate till stop-1. The range(1,5) will generate numbers 1 to 4 iterations. It is optional.
* The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

Example-1: Program to print numbers in sequence.

for i in range(10):

print(i,end = ' ')

Output:

0 1 2 3 4 5 6 7 8 9

Example - 2: Program to print table of given number.

n = int(input("Enter the number "))

for i in range(1,11):

c = n\*i

print(n,"\*",i,"=",c)

Output:

Enter the number 10

10 \* 1 = 10

10 \* 2 = 20

10 \* 3 = 30

10 \* 4 = 40

10 \* 5 = 50

10 \* 6 = 60

10 \* 7 = 70

10 \* 8 = 80

10 \* 9 = 90

10 \* 10 = 100

Example-3: Program to print even number using step size in range().

n = int(input("Enter the number "))

for i in range(2,n,2):

print(i)

Output:

Enter the number 20

2

4

6

8

10

12

14

16

18

We can also use the range() function with sequence of numbers. The len() function is combined with range() function which iterate through a sequence using indexing. Consider the following example.

str=”PYTHON”

for i in range(len(str)):

print(str[i])

Output:

P

Y

T

H

O

N

**Nested for loop in python**

Python allows us to nest any number of for loops inside a for loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

Syntax:

for iterating\_var1 in sequence: #outer loop

for iterating\_var2 in sequence: #inner loop

#block of statements

#Other statements

Example- 1: Nested for loop

rows = int(input("Enter the rows:"))

for i in range(0,rows+1):

for j in range(i):

print("\*",end = '')

print()

Output:

Enter the rows:5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

Example-2 : without Nested for loop

rows = int(input("Enter the rows:"))

for i in range(1,rows+1):

print("\*"\*i)

Output:

Enter the rows:5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

**Using else statement with for loop**

Unlike other languages like C, C++, or Java, Python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

Example 1

for i in range(0,5):

print(i)

else:

print("for loop completely exhausted, since there is no break.")

Output:

0

1

2

3

4

for loop completely exhausted, since there is no break.

Example 2

for i in range(0,5):

print(i)

break;

else:print("for loop is exhausted");

print("The loop is broken due to break statement...came out of the loop")

Output:

0

The loop is broken due to break statement...came out of the loop

**Python While loop:**

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.

It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

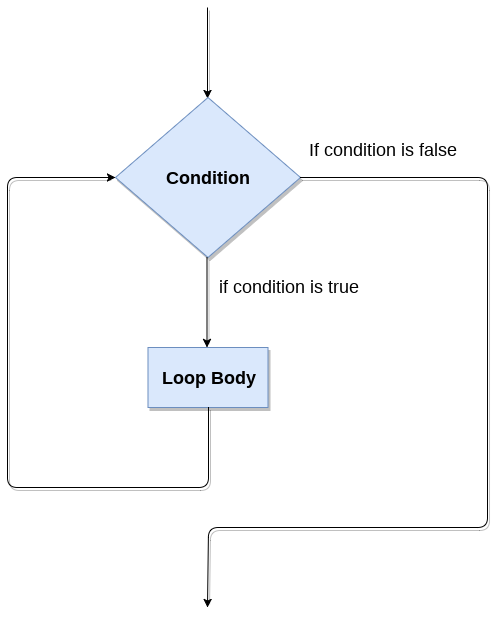
Syntax:

while expression:

statements

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

While loop Flowchart



Example-1: Program to print 1 to 10 using while loop

i=1

while(i<=10):

print(i)

i=i+1

Output:

1

2

3

4

5

6

7

8

9

10

Example -2: Program to print table of given numbers.

i=1

number = int(input("Enter the number:"))

while i<=10:

print("%d X %d = %d \n"%(number,i,number\*i))

i = i+1

Output:

Enter the number:10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

**Infinite while loop**

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Example 1

while (1):

print("Hi! we are inside the infinite while loop")

Output:

Hi! we are inside the infinite while loop

Hi! we are inside the infinite while loop

Example 2

var = 1

while(var != 2):

i = int(input("Enter the number:"))

print("Entered value is %d"%(i))

Output:

Enter the number:10

Entered value is 10

Enter the number:10

Entered value is 10

Enter the number:10

Entered value is 10

Infinite time

Example:

while (1):

print("------Menu-----")

print("1.Adddition")

print("2.Sub")

print("3.Mult")

print("4.Exit")

ch=int(input("Enter option:"))

if ch==1:

num1=int(input("Enter 1st val:"))

num2=int(input("Enter 1st val:"))

num3=num1+num2

print("result: ",num3)

elif ch==2:

num1=int(input("Enter 1st val:"))

num2=int(input("Enter 1st val:"))

num3=num1-num2

print("result: ",num3)

elif ch==3:

num1=int(input("Enter 1st val:"))

num2=int(input("Enter 1st val:"))

num3=num1\*num2

print("result: ",num3)

elif ch==4:

break

else:

print("Enter correct choice")

**Using else with while loop**

Python allows us to use the else statement with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed, and the statement present after else block will be executed. The else statement is optional to use with the while loop.

Example 1

i=1

while(i<=5):

print(i)

i=i+1

else:

print("The while loop exhausted")

Example 2

i=1

while(i<=5):

print(i)

i=i+1

if(i==3):

break

else:

print("The while loop exhausted")

Output:

1

2

In the above code, when the break statement encountered, then while loop stopped its execution and skipped the else statement.

Example-3 Program to print Fibonacci numbers to given limit

terms = int(input("Enter the terms "))

# first two intial terms

a = 0

b = 1

count = 0

if (terms <= 0):

print("Please enter a valid integer")

elif (terms == 1):

print("Fibonacci sequence upto",limit,":")

print(a)

else:

print("Fibonacci sequence:")

while (count < terms) :

print(a, end = ' ')

c = a + b

a = b

b = c

count += 1

Output:

Enter the terms 10

Fibonacci sequence:

0 1 1 2 3 5 8 13 21 34

Example: Use while as do while

i=1

while (1):

print(i)

i=i+1

if i>5:

break

**Loop Control Statements:**

1. **break**
2. **continue**
3. **pass**

**Python break statement**

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break :

#loop statements

break;

Example 1

list =[1,2,3,4]

count = 1;

for i in list:

if i == 4:

print("item matched")

count = count + 1;

break

print("found at",count,"location");

Output:

item matched

found at 2 location

Example 2

str = "python"

for i in str:

if i == 'o':

break

print(i);

Output:

p

y

t

h

Example 3: break statement with while loop

i = 0;

while 1:

print(i," ",end=""),

i=i+1;

if i == 10:

break;

print("came out of while loop");

Output:

0 1 2 3 4 5 6 7 8 9 came out of while loop

Example 3

n=2

while 1:

i=1;

while i<=10:

print("%d X %d = %d\n"%(n,i,n\*i));

i = i+1;

choice = int(input("Do you want to continue printing the table, press 0 for no?"))

if choice == 0:

break;

n=n+1

Output:

2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1

3 X 1 = 3

3 X 2 = 6

3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0

**Python continue Statement**

The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition. The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

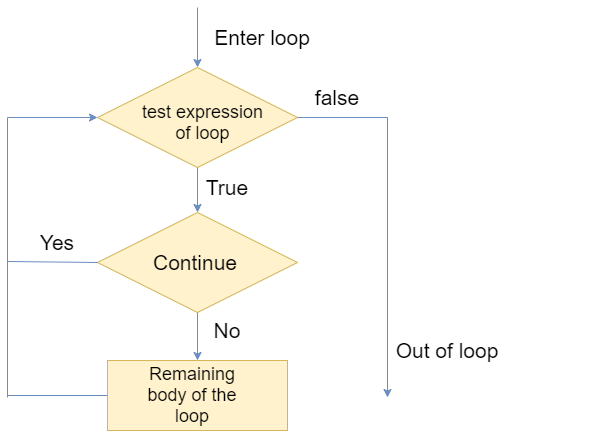
Syntax

#loop statements

continue

#the code to be skipped

Flow Diagram



Example 1

i = 0

while(i < 10):

i = i+1

if(i == 5):

continue

print(i)

Output:

1

2

3

4

6

7

8

9

10

Example 2

str = "Python"

for i in str:

if(i == 't'):

continue

print(i)

Output:

P

Y

H

O

N

**Python Pass**

In Python, the pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty. It is the same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.

It is beneficial when a statement is required syntactically, but we want we don't want to execute or execute it later. The difference between the comments and pass is that, comments are entirely ignored by the Python interpreter, where the pass statement is not ignored.

Suppose we have a loop, and we do not want to execute right this moment, but we will execute in the future. Here we can use the pass.

Example - Pass statement

# pass is just a placeholder we will add functionality later.

values = {'P', 'y', 't', 'h','o','n'}

for val in values:

pass

Example - 2:

for i in [1,2,3,4,5]:

if(i==4):

pass

print("This is pass block",i)

print(i)

Output:

1

2

3

This is pass block 4

4

5

**Note: We can create empty class or function using the pass statement.**

# Empty Function

def function\_name(args):

pass

#Empty Class

class Python:

pass

**Python String**

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.

Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Syntax:

str = "Hi Python !"

Here, if we check the type of the variable str using a Python script print(type(str)), then it will print a string (str).

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

**Creating String in Python**

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or docstrings.

#Using single quotes

str1 = 'Hello Python'

print(str1)

#Using double quotes

str2 = "Hello Python"

print(str2)

#Using triple quotes

str3 = '''''Triple quotes are generally used for

represent the multiline or

docstring”””

print(str3)

Output:

Hello Python

Hello Python

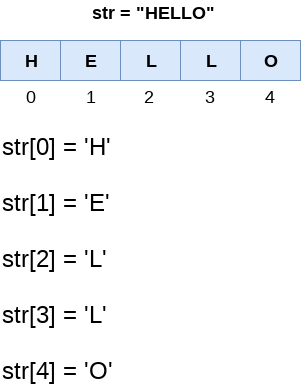
Triple quotes are generally used for

represent the multiline or

docstring

**Strings indexing and splitting**

Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



Example:

string1='test data1'

string2="test data2"

string3='''test \

data3'''

string4="""test

data4"""

print(string1)

print(string2)

print(string3)

print(string4)

print(string1[0])

print(string1[5])

print(string1[-1])

for i in range(-1,-11,-1):

print(string1[i])

Example:

str = "HELLO"

print(str[0])

print(str[1])

print(str[2])

print(str[3])

print(str[4])

# It returns the IndexError because 6th index doesn't exist

print(str[6])

Output:

H

E

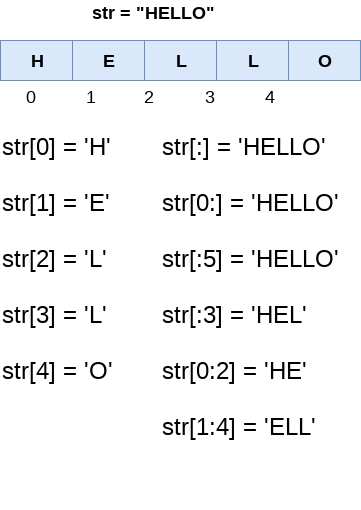
L

L

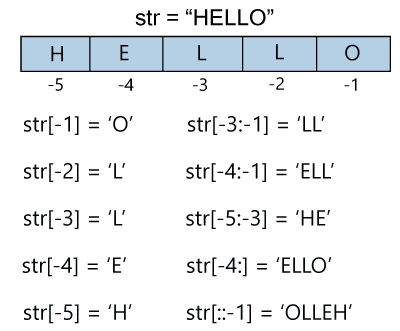
O

IndexError: string index out of range

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the : (colon) operator in Python to access the substring from the given string.



We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on. Consider the following image.



**Reassigning Strings**

Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. **Strings are immutable in Python.**

Example 1

str = "HELLO"

str[0] = "h"

print(str)

Output:

Traceback (most recent call last):

File "12.py", line 2, in <module>

str[0] = "h";

TypeError: 'str' object does not support item assignment

However, in example 2, the string str can be assigned completely to a new content as specified in the following example.

Example 2

str = "HELLO"

print(str)

str = "hello"

print(str)

Output:

HELLO

hello

**Deleting the String**

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the del keyword.

str = "Python"

del str[1]

Output:

TypeError: 'str' object doesn't support item deletion

Now we are deleting entire string.

str1 = "Python"

del str1

print(str1)

Output:

NameError: name 'str1' is not defined

**String Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | It is known as **concatenation operator** used to join the strings given either side of the operator. |
| \* | It is known as **repetition operator**. It concatenates the multiple copies of the same string. |
| [] | It is known as **slice operator**. It is used to access the sub-strings of a particular string. |
| [:] | It is known as **range slice operator**. It is used to access the characters from the specified range. |
| in | It is known as **membership operator**. It returns if a particular sub-string is present in the specified string. |
| not in | It is also a **membership operator** and does the exact reverse of in. It returns true if a particular substring is not present in the specified string. |
| r/R | It is used to specify the **raw string**. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string. |
| % | It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python. |

**Example**

Consider the following example to understand the real use of Python operators.

str = "Hello"

str1 = " world"

print(str\*3) # prints HelloHelloHello

print(str+str1) # prints Hello world

print(str[4]) # prints o

print(str[2:4]); # prints ll

print('w' in str) # prints false as w is not present in str

print('wo' not in str1) # prints false as wo is present in str1.

print(r'C:\\python') # prints C:\\python as it is written

print("The string str : %s"%(str)) # prints The string str : Hello

Output:

HelloHelloHello

Hello world

o

ll

False

False

C:\\python

The string str : Hello

Example:

string1='test data1'

string2="test data2"

string3='''test \

data3'''

num=5

print("The string str : %s and value of num=%d %s"%(string1,num,string2))

print("The string str : ",string1,"and value of num=",num)

print(f"The string str : {string1} and value of num={num} {string2}")

Output:

The string str : test data1 and value of num=5 test data2

The string str : test data1 and value of num= 5

The string str : test data1 and value of num=5 test data2

**Python String Formatting**

**Escape Sequence**

Let's suppose we need to write the text as - They said, "Hello what's going on?"- the given statement can be written in single quotes or double quotes but it will raise the SyntaxError as it contains both single and double-quotes.

**Example**

Consider the following example to understand the real use of Python operators.

str = "They said, "Hello what's going on?""

print(str)

Output:

SyntaxError: invalid syntax

We can use the triple quotes to accomplish this problem but Python provides the escape sequence.

The backslash(\) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently. The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

**Example -**

# using triple quotes

print('''''They said, "What's there?"''')

# escaping single quotes

print('They said, "What\'s going on?"')

# escaping double quotes

print("They said, \"What's going on?\"")

Output:

They said, "What's there?"

They said, "What's going on?"

They said, "What's going on?"

**The list of an escape sequence is given below:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Escape Sequence** | **Description** | **Example** |
| 1. | \ | It ignores the new line. | print("Python1 \  Python2 \  Python3")  Output:  Python1 Python2 Python3 |
| 2. | \\ | Backslash | print("\\")  Output:  \ |
| 3. | \' | Single Quotes | print('\'')  Output:  ' |
| 4. | \'' | Double Quotes | print("\"")  Output:  " |
| 5. | \a | ASCII Bell | print("\a") |
| 6. | \b | ASCII Backspace(BS) | print("Hello\bWorld")  Output:  Hello World |
| 8. | \n | ASCII Linefeed | print("Hello \n World!")  Output:  Hello  World! |
| 9. | \r | ASCII Carriege Return(CR) | print("Hello \r World!")  Output:  World! |
| 10. | \t | ASCII Horizontal Tab | print("Hello \t World!")  Output:  Hello World! |
| 11. | \v | ASCII Vertical Tab | print("Hello \v World!")  Output:  Hello  World! |
| 12. | \ooo | Character with octal value | print("\110\145\154\154\157")  Output:  Hello |
| 13 | \xHH | Character with hex value. | print("\x48\x65\x6c\x6c\x6f")  Output:  Hello |

**Example:**

print("C:\\Users\\DEVANSH SHARMA\\Python32\\Lib")

print("This is the \n multiline quotes")

print("This is \x48\x45\x58 representation")

Output:

C:\Users\DEVANSH SHARMA\Python32\Lib

This is the

multiline quotes

This is HEX representation

**We can ignore the escape sequence from the given string by using the raw string. We can do this by writing r or R in front of the string.**

Example:

print(r"C:\\Users\\test\\Python")

Output:

C:\\Users\\test\\Python

**The format() method**

The format() method is the most flexible and useful method in formatting strings. **The curly braces {} are used as the placeholder in the string and replaced by the format() method argument**.

Example:

print("{} and {} both are the best friend".format("Devansh","Abhishek"))

print("{1} and {0} best players ".format("Virat","Rohit"))

#Keyword Argument

print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))

Output:

Devansh and Abhishek both are the best friend

Rohit and Virat best players

James,Peter,Ricky

**Python String Formatting Using % Operator**

Python allows us to use the format specifiers used in C's printf statement. The format specifiers in Python are treated in the same way as they are treated in C. However, Python provides an additional operator %, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

Example:

Integer = 10;

Float = 1.290

String = "Devansh"

print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%(Integer,Float,String))

Output:

Hi I am Integer ... My value is 10

Hi I am float ... My value is 1.290000

Hi I am string ... My value is Devansh

**Python String functions**

**Python provides various in-built functions that are used for string handling. Many String functions**

|  |  |
| --- | --- |
| Method | Description |
| capitalize() | It capitalizes the first character of the String.  string="test data"  string.capitalize()  print(string.capitalize())  output: Test data |
| casefold() | It returns a version of s suitable for case-less comparisons.  string="TeSt Data"  print(string.casefold())  Output: test data |
| center(width ,fillchar) | It returns a space padded string with the original string centred with equal number of left and right spaces.  string="TeSt Data"  print(string.center(20 ,'\*'))  Output: \*\*\*\*\*TeSt Data\*\*\*\*\*\* |
| count(string,begin,end) | It counts the number of occurrences of a substring in a String between begin and end index. string="TeSt Data"  print(string.center(20 ,'\*'))  string="teSt Date"  print(string.count('te',0,10))  Output: 2 |
| endswith(suffix ,begin=0,end=len(string)) | It returns a Boolean value if the string terminates with given suffix between begin and end. |
| find(substring ,beginIndex, endIndex) | It returns the index value of the string where substring is found between begin index and end index. |
| format(value) | It returns a formatted version of S, using the passed value.  str = "Java"  str2 = "C#"  str3 = "{} and {} both are programming languages".format(str,str2)  print(str3) |
| index(subsring, beginIndex, endIndex) | It throws an exception if string is not found. It works same as find() method. |
| isalnum() | It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false. |
| isalpha() | It returns true if all the characters are alphabets and there is at least one character, otherwise False. |
| isdecimal() | It returns true if all the characters of the string are decimals.  str = "123" # True  str3 = "2.50" # False  str2 = str.isdecimal()  str4 = str3.isdecimal()  print(str2)  print(str4)  Output:  True  False |
| isdigit() | It returns true if all the characters are digits and there is at least one character, otherwise False.  str = "12345"  str3 = "120-2569-854"  str2 = str.isdigit()  str4 = str3.isdigit()  print(str2)  print(str4)  Output:  True  False  c = '\u00B2'  print( c.isdecimal())  print( c.isdigit())  Output:  False  True |
| isnumeric() | It returns true if the string contains only numeric characters.  >>> c = '\u00BD' # ½  >>> c.isdecimal()  False  >>> c.isdigit()  False  >>> c.isnumeric()  True |
| isidentifier() | It returns true if the string is the valid identifier. |
| islower() | It returns true if the characters of a string are in lower case, otherwise false. |
| isupper() | It returns false if characters of a string are in Upper case, otherwise False. |
| isspace() | It returns true if the characters of a string are white-space, otherwise false. |
| istitle() | It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case. |
| join(seq) | It merges the strings representation of the given sequence.  str = "->" # string  list = {'Java','C#','Python'} # iterable  str2 = str.join(list)  print(str2)  Output: Java->Python->C# |
| len(string) | It returns the length of a string. |
| ljust(width[,fillchar]) rjust(width[,fillchar]) | It returns the space padded strings with the original string left justified to the given width. |
| lower()  upper() | To convert all characters are in small case. Ex: str.lower() |
| lstrip([char])  rstrip([char]) | It removes all leading whitespaces of a string and can also be used to remove particular character from leading. |

**Python List**

A list in Python is used to store the sequence of various types of data. Python lists are **mutable type** its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below

L1 = ["John", 102, "USA"]

L2 = [1, 2, 3, 4, 5, 6]

If we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

print(type(L1))

print(type(L2))

Output:

<class 'list'>

<class 'list'>

**Characteristics of Lists**

* The lists are ordered.
* The element of the list can access by index.
* The lists are mutable types.
* A list can store the number of various elements.

**Lists are the ordered.**

a = [1,2,"Peter",4.50,"Ricky",5,6]

b = [1,2,5,"Peter",4.50,"Ricky",6]

print(a ==b)

Output:

False

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

a = [1, 2,"Peter", 4.50,"Ricky",5, 6]

b = [1, 2,"Peter", 4.50,"Ricky",5, 6]

print(a == b)

Output:

True

Example:

emp = ["John", 102, "USA"]

Dep1 = ["CS",10]

Dep2 = ["IT",11]

HOD\_CS = [10,"Mr. Holding"]

HOD\_IT = [11, "Mr. Bewon"]

print("printing employee data...")

print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))

print("printing departments...")

print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))

print("HOD Details ....")

print("CS HOD Name: %s, Id: %d"%(HOD\_CS[1],HOD\_CS[0]))

print("IT HOD Name: %s, Id: %d"%(HOD\_IT[1],HOD\_IT[0]))

print(type(emp),type(Dep1),type(Dep2),type(HOD\_CS),type(HOD\_IT))

Output:

printing employee data...

Name : John, ID: 102, Country: USA

printing departments...

Department 1:

Name: CS, ID: 11

Department 2:

Name: IT, ID: 11

HOD Details ....

CS HOD Name: Mr. Holding, Id: 10

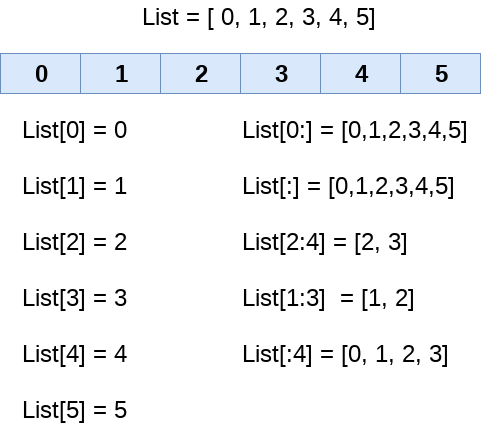
IT HOD Name: Mr. Bewon, Id: 11

<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>

**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**

**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

Example:

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

print(list[0])

print(list[1])

print(list[2])

print(list[3])

print(list[0:6])

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]

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.

Python Lists

Example:

list = [1,2,3,4,5]

print(list[-1])

print(list[-3:])

print(list[:-1])

print(list[-3:-1])

Output:

5

[3, 4, 5]

[1, 2, 3, 4]

[3, 4]

**Iterating a List**

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

Example:

list = ["John", "David", "James", "Jonathan"]

for i in list:

# The i variable will iterate over the elements of the List and contains each element in each iteration.

print(i)

Output:

John

David

James

Jonathan

**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.

Example:

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

print(list)

list[2] = 10

print(list)

# Adding multiple-element

list[1:3] = [89, 78]

print(list)

# It will add value at the end of the list

list[-1] = 25

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]

**Adding elements to the list using append and insert method:**

Python provides append() function which is used to add an element to the list. However, the **append() function can only add value to the end of the list**.

Example:

#Declaring the empty list

l =[]

n = int(input("Enter the number of elements in the list:"))

for i in range(0,n):

**l.append(input("Enter the item:"))**

print("printing the list items..")

for i in l:

print(i, end = " ")

Output:

Enter the number of elements in the list:5

Enter the item:25

Enter the item:46

Enter the item:12

Enter the item:75

Enter the item:42

printing the list items

25 46 12 75 42

Example:

listtest=[12,"Rishabh",99.8]

**listtest.append("Kisalaya is a good boy")**

print(listtest)

**listtest.insert(2," Rishabh is a good boy")**

print(listtest)

Output:

[12, 'Rishabh', 99.8, 'Kisalaya is a good boy']

[12, 'Rishabh', 'Rishabh is a good boy', 99.8, 'Kisalaya is a good boy']

**Deleting/Removing elements from list:**

The list elements can also be deleted by **using the del keyword**. Python also provides us the **remove() method** to delete the element.

Note: **del can delete one or more elements using index number. Remove method take value(only one) as input to delete element.**

Example:

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

print(list)

del list[2]

print(list)

del list[1:3]

print(list)

del list[-1]

print(list)

Output:

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

[1, 2, 4, 5, 6]

[1, 5, 6]

[1, 5]

**Removing elements using from the list**

Python provides the remove() function which is used to remove the element from the list.

Example -

list = [0,1,2,3,4]

print("printing original list: ");

for i in list:

print(i,end=" ")

list.remove(2)

print("\nprinting the list after the removal of first element...")

for i in list:

print(i,end=" ")

Output:

printing original list:

0 1 2 3 4

printing the list after the removal of first element...

0 1 3 4

Example:

listtest=[12,"Rishabh",99.8]

listtest.append("Kishlay is a good boy")

print(listtest)

listtest.insert(2,"Rishav is a good boy")

print(listtest)

listtest.remove(12)

print(listtest)

del listtest[0]

print(listtest)

Output:

[12, 'Rishabh', 99.8, 'Kishlay is a good boy']

[12, 'Rishabh', 'Rishav is a good boy', 99.8, 'Kishlay is a good boy']

['Rishabh', 'Rishav is a good boy', 99.8, 'Kishlay is a good boy']

['Rishav is a good boy', 99.8, 'Kishlay is a good boy']

**Remove all elements from list:**

We can use **clear()** method to remove all elements from list.

Example:

list = ["Python","abc"]

list.clear()

print(list)

Output:

[]

**Python List Operations**

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

Consider a Lists l1 = [1, 2, 3, 4], and l2 = [5, 6, 7, 8] to perform operation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Repetition | The repetition operator enables the list elements to be repeated multiple times. | L1\*2 = [1, 2, 3, 4, 1, 2, 3, 4] |
| Concatenation | It concatenates the list mentioned on either side of the operator. | l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8] |
| Membership | It returns true if a particular item exists in a particular list otherwise false. | print(2 in l1) prints True. |
| Iteration | The for loop is used to iterate over the list elements. | for i in l1:  print(i,end=” “)  Output  1 2 3 4 |
| Length | It is used to get the length of the list | len(l1) = 4 |

**Python List Built-in functions**

Python provides the following built-in functions, which can be used with the lists.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Function** | **Description** | **Example** |
| 1 | len(list) | It is used to calculate the length of the list. | L1 = [1,2,3,4,5,6,7,8]  print(len(L1))  8 |
| 2 | max(list) | It returns the maximum element of the list. | L1 = [12,34,26,48,72]  print(max(L1))  72 |
| 3 | min(list) | It returns the minimum element of the list. | L1 = [12,34,26,48,72]  print(min(L1))  12 |
| 4 | list(seq) | It converts any sequence to the list. | str = "Johnson"  s = list(str)  print(type(s))  <class list> |

Example: 1- Write the program to remove the duplicate element of the list.

list1 = [1,2,2,3,55,98,65,65,13,29]

list2 = []

for i in list1:

if i not in list2:

list2.append(i)

print(list2)

Output:

[1, 2, 3, 55, 98, 65, 13, 29]

Example:2- Write a program to find the sum of the element in the list.

list1 = [3,4,5,9,10,12,24]

sum = 0

for i in list1:

sum = sum+i

print("The sum is:",sum)

Output:

The sum is: 67

Example: 3- Write the program to find the lists consist of at least one common element.

list1 = [1,2,3,4,5,6]

list2 = [7,8,9,2,10]

for x in list1:

for y in list2:

if x == y:

print("The common element is:",x)

Output:

The common element is: 2

**Python Tuple**

Python Tuple is used to store the sequence of **immutable** Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the **tuple is immutable, and the value of the items stored in the tuple cannot be changed.**

**Creating a tuple**

A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The **parentheses are optional but it is good practice to use**. A tuple can be defined as follows.

T1 = (101, "Peter", 22)

T2 = ("Apple", "Banana", "Orange")

T3 = 10,20,30,40,50

print(type(T1))

print(type(T2))

print(type(T3))

Output:

<class 'tuple'>

<class 'tuple'>

<class 'tuple'>

**An empty tuple can be created as follows.**

T4 = ()

**Creating a tuple with single element is slightly different. We will need to put comma after the element to declare the tuple.**

tup1 = ("Python")

print(type(tup1))

#Creating a tuple with single element

tup2 = ("Python",)

print(type(tup2))

Output:

<class 'str'>

<class 'tuple'>

**A tuple is indexed in the same way as the lists.** The items in the tuple can be accessed by using their specific index value.

Example -

tuple1 = (10, 20, 30, 40, 50, 60)

print(tuple1)

count = 0

for i in tuple1:

print("tuple1[%d] = %d"%(count, i))

count = count+1

Output:

(10, 20, 30, 40, 50, 60)

tuple1[0] = 10

tuple1[1] = 20

tuple1[2] = 30

tuple1[3] = 40

tuple1[4] = 50

tuple1[5] = 60

**Accept elements of tuple from user:**

Example -

tuple1 = tuple(input("Enter the tuple elements ..."))

print(tuple1)

count = 0

for i in tuple1:

print("tuple1[%d] = %s"%(count, i))

count = count+1

Output:

Enter the tuple elements ...123456

('1', '2', '3', '4', '5', '6')

tuple1[0] = 1

tuple1[1] = 2

tuple1[2] = 3

tuple1[3] = 4

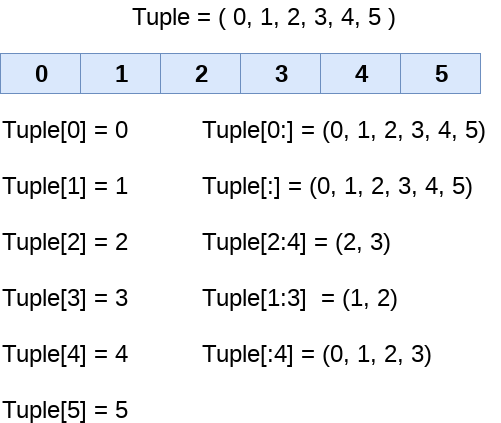
tuple1[4] = 5

tuple1[5] = 6

**Tuple indexing and slicing**

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.



Example:

tup = (1,2,3,4,5,6,7)

print(tup[0])

print(tup[1])

print(tup[2])

# It will give the IndexError

print(tup[8])

Output:

1

2

3

tuple index out of range

In the above code, the tuple has 7 elements which denote 0 to 6. We tried to access an element outside of tuple that raised an IndexError.

**Example:**

tuple = (1,2,3,4,5,6,7)

#element 1 to end

print(tuple[1:])

#element 0 to 3 element

print(tuple[:4])

#element 1 to 4 element

print(tuple[1:5])

# element 0 to 6 and take step of 2

print(tuple[0:6:2])

Output:

(2, 3, 4, 5, 6, 7)

(1, 2, 3, 4)

(1, 2, 3, 4)

(1, 3, 5)

**Negative Indexing**

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on.

The elements from left to right are traversed using the negative indexing.

Example:

tuple1 = (1, 2, 3, 4, 5)

print(tuple1[-1])

print(tuple1[-4])

print(tuple1[-3:-1])

print(tuple1[:-1])

print(tuple1[-2:])

Output:

5

2

(3, 4)

(1, 2, 3, 4)

(4, 5)

**Deleting Tuple**

Unlike lists, the **tuple items cannot be deleted by using the del keyword** **as tuples are immutable**. To delete an entire tuple, we can use the del keyword with the tuple name.

Example:

tuple1 = (1, 2, 3, 4, 5, 6)

print(tuple1)

del tuple1[0]

print(tuple1)

del tuple1

print(tuple1)

Output:

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

Traceback (most recent call last):

File "tuple.py", line 4, in <module>

print(tuple1)

NameError: name 'tuple1' is not defined

**Basic Tuple operations**

The operators like concatenation (+), repetition (\*), Membership (in) works in the same way as they work with the list.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| Repetition | The repetition operator enables the tuple elements to be repeated multiple times. | T1\*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5) |
| Concatenation | It concatenates the tuple mentioned on either side of the operator. | T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9) |
| Membership | It returns true if a particular item exists in the tuple otherwise false | print (2 in T1) prints True. |
| Iteration | The for loop is used to iterate over the tuple elements. | for i in T1:  print(i,end=” “)  Output  1 2 3 4 5 |
| Length | It is used to get the length of the tuple. | len(T1) = 5 |

**Python Tuple built-in functions:**

|  |  |  |
| --- | --- | --- |
| SN | Function | Description |
| 1 | len(tuple) | It calculates the length of the tuple. |
| 2 | max(tuple) | It returns the maximum element of the tuple |
| 3 | min(tuple) | It returns the minimum element of the tuple. |
|  |  |  |
| 4 | tuple(seq) | It converts the specified sequence to the tuple. |

**Where use tuple?**

Using tuple instead of list is used in the following scenario.

* Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.
* Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.

**List vs. Tuple**

|  |  |  |
| --- | --- | --- |
| SN | List | Tuple |
| 1 | The literal syntax of list is shown by the []. | The literal syntax of the tuple is shown by the (). |
| 2 | The List is mutable. | The tuple is immutable. |
| 3 | The List has a variable length. | The tuple has the fixed length. |
| 4 | The list provides more functionality than a tuple. | The tuple provides less functionality than the list. |
| 5 | The list is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed. | The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items cannot be changed. It can be used as the key inside the dictionary. |
| 6 | The lists are less memory efficient than a tuple. | The tuples are more memory efficient because of its immutability. |

**Main differences in the following points:**

**Representation Differences**

The representation of the Lists and tuple is marginally different. List are commonly enclosed with the square bracket [], and elements are comma-separated element. Tuples are enclosed with parenthesis (), and elements are separated by the comma. The parenthesis is optional to use, and these types of tuples are called tuple packing.

Example:

list1 = ['Python', 1, 2, 54.30, {'Name: ''Peter'}]

print(type(list))

tuple1 = ('Python',5,8,31.9,[1,2,3])

print(type(tuple1))

Output:

<class 'list'>

<class 'tuple'>

**Mutable Lists and Immutable Tuples**

It is the most important difference between list and tuple whereas lists are mutable, and tuples are immutable. The lists are mutable which means the Python object can be modified after creation, whereas tuples can't be modified after creation.

Example:

a = ["Peter","Joseph","Mathew","Ricky"]

print(a)

Output:

['Peter', 'Joseph', 'Mathew', 'Ricky']

Now we are changing 0th index element "Peter" to "Samson".

a[0] = "Samson"

print(a)

Output:

['Samson', 'Joseph', 'Mathew', 'Ricky']

Now we create a tuple and do the same thing.

a = (10,20,"Python",30,40)

print(a)

Output:

(10, 20, 'Python', 30, 40)

a[0] = 50

Output:

TypeError Traceback (most recent call last)

<ipython-input-5-52b2981fae12> in <module>

----> 1 a[0] = 50

TypeError: 'tuple' object does not support item assignment

**Debugging**

The tuples are easy to debug in a big project because of its immutability. If we have a small project or less number of data, then lists play an effective role.

Example:

a = [6,9,4,3,7,0,1]

# Copying address of a in b

b = a

a[3] = "Python"

print(a)

Output:

[6, 9, 4, 'Python', 7, 0, 1]

In the above code, we did b = a; here we are not copying the list object from b to a. The b referred to the address of the list a. It means if we make the change in the b then that will reflect the same as in list a, and it makes debugging easy. But it is hard for the significant project where Python objects may have multiple references.

It will be very complicated to track those changes in lists but immutable object tuple can't change after created.

So tuples are easy to debug.

**Functions Support**

The tuples support less operation than the list. The inbuilt dir(object) is used to get all the supported functions for the list and tuple.

**List Functions**

dir(list)

Output:

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

**Tuple Functions**

dir(tuple)

Output:

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'count', 'index']

**Memory Efficient**

The tuples are more memory efficient than the list because tuple has less built-in operations. Lists are suitable for the fewer elements whereas tuples are a bit faster than the list for the huge amount of data.

Example:

Tuple = (1,2,3,4,5,6,7,8,9,0,5485,87525,955,3343,53234,6423,623456,234535)

List = [1,2,3,4,5,6,7,8,9,0,78,34,43,32,43,55,54,212,642,533,43434,54532 ]

print('Tuple size =', Tuple.\_\_sizeof\_\_()) # Tuple size = 52

print('List size =', List.\_\_sizeof\_\_())

Output:

Tuple size = 168

List size = 216

**Python Set**

A Python set is the **collection of the unordered items**. Each element in the **set must be unique, mutable**, 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.

**Creating a set**

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example 1: Using curly braces

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

print(Days)

print(type(Days))

print("looping through the set elements :")

for i in Days:

print(i)

Output:

{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

<class 'set'>

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday

Example 2: Using set() method

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

print(Days)

print(type(Days))

print("looping through the set elements ... ")

for i in Days:

print(i)

Output:

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

<class 'set'>

looping through the set elements ...

Friday

Wednesday

Thursday

Saturday

Monday

Tuesday

Sunday

**Note: It can contain any type of element such as integer, float, tuple etc.** **But mutable elements (list, dictionary, set) can't be a member of set**.

# Creating a set which have immutable elements

set1 = {1,2,3, "Python", 20.5, 14}

print(type(set1))

#Creating a set which have mutable element

set2 = {1,2,3,["Python",4]}

print(type(set2))

Output:

<class 'set'>

Traceback (most recent call last)

<ipython-input-5-9605bb6fbc68> in <module>

4

5 #Creating a set which holds mutable elements

----> 6 set2 = {1,2,3,[" Python",4]}

7 print(type(set2))

TypeError: unhashable type: 'list'

In the above code, we have created two sets, the set set1 have immutable elements and set2 have one mutable element as a list. While checking the type of set2, it raised an error, which means set can contain only immutable elements.

**Empty set:**

**Creating an empty set is a bit different because empty curly {} braces are also used to create a dictionary as well. So Python provides the set() method used without an argument to create an empty set.**

# Empty curly braces will create dictionary

set3 = {}

print(type(set3))

# Empty set using set() function

set4 = set()

print(type(set4))

Output:

<class 'dict'>

<class 'set'>

**Let's see what happened if we provide the duplicate element to the set.**

set5 = {1,2,4,4,5,8,9,9,10}

print("Return set with unique elements:",set5)

Output:

Return set with unique elements: {1, 2, 4, 5, 8, 9, 10}

In the above code, we can see that set5 consisted of multiple duplicate elements when we printed it remove the duplicity from the set.

**Adding items to the set**

Python provides the add() method and update() method which can be used to add some particular item to the set. The **add() method is used to add a single** element whereas the **update() method is used to add multiple elements to the set**.

**add() Method**

Python add() method adds new element to the set. It takes a parameter, the element to add. It returns None to the caller.

Syntax: add(elem)

Parameters: elem: element to be added.

Return: It returns None.

Example 1: A simple example to add a new element in the set. It returns a modified set.

set = {1,2,3}

print(set)

set.add(4)

print("After adding new element: \n",set)

Output:

{1, 2, 3}

After adding new element:

{1, 2, 3, 4}

Example 2: Adding an element which already exist will not modifiy the set. Set does not store duplicate elements.

set = {1,2,3}

print(set)

set.add(2)

print("After adding new element: \n",set)

Output:

{1, 2, 3}

After adding new element:

{1, 2, 3}

Example 3: Set also allows to store other data structures like tuple, list etc.

set = {1,2,3}

print(set)

tup = (4,5)

set.add(tup)

print("After adding new element: \n",set)

tup1 = (2,3,4)

set.add(tup1)

print("After adding tuple: \n",set)

Output:

{1, 2, 3}

After adding new element:

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

After adding tuple:

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

Example: 1 - Using add() method

Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nAdding other months to the set...");

Months.add("July");

Months.add ("August");

print("\nPrinting the modified set...");

print(Months)

print("\nlooping through the set elements ... ")

for i in Months:

print(i)

Output:

printing the original set ...

{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...

{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...

February

July

May

April

March

August

June

January

**To add more than one item in the set, Python provides the update() method.** It accepts iterable as an argument.

Example - 2 Using update() function

Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nupdating the original set ... ")

Months.update(["July","August","September","October"]);

print("\nprinting the modified set ... ")

print(Months);

Output:

printing the original set ...

{'January', 'February', 'April', 'May', 'June', 'March'}

updating the original set ...

printing the modified set ...

{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}

**Removing items from the set**

Python provides **the discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these function, **using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error**.

**Python Set discard() Method**

Python discard() method discards or remove the elememt from the set. This method does not return anything, even no error if the elememt is not present. It takes a parameter which is an elememt to be removed.

Syntax: discard(elem)

Parameters: elem: element to be deleted.

Return: It returns None.

Example 1:A simple example to use discard method to remove an element.

set = {1,2,3,4,5}

print(set)

set.discard(2)

print(set)

Output:

{1, 2, 3, 4, 5}

{1, 3, 4, 5}

Example 2:If the element is not present it returns none to the caller method.

set = {1,2,3,4,5}

print(set)

val = set.discard(22)

print(val)

Output:

{1, 2, 3, 4, 5}

None

Example 3: An example where we are implementing this method into a program. It removes all odds elements.

set = {1,2,3,4,5}

set2 = {1,2,3,4,5}

print(set)

for s in set2:

if s%2!=0:

set.discard(s)

print(set)

Output:

{1, 2, 3, 4, 5}

{2, 4}

Example-1 **Using discard() method**

months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(months)

print("\nRemoving some months from the set...");

months.discard("January");

months.discard("May");

print("\nPrinting the modified set...");

print(months)

print("\nlooping through the set elements ... ")

for i in months:

print(i)

Output:

printing the original set ...

{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...

{'February', 'March', 'April', 'June'}

looping through the set elements ...

February

March

April

June

**Python Set remove() Method**

Python remove() method removes an element elem from the set. It raises error KeyError if elem is not contained in the set.

Syntax: remove(elem)

Parameters: elem: element to be deleted.

Return: It returns None but throws KeyError if the value does not found in the set.

Example 1: Let's first see a simple example to remove an element from the set.

set = {1,2,3}

print(set)

set.remove(1)

print("After removing element: \n",set)

Output:

{1, 2, 3}

After removing element:

{2, 3}

Example 2: It throws an error KeyError if the element is not available in the set.

set = {1,2,3}

print(set)

set.remove(22)

print("After removing element: \n",set)

Output:

set.remove(22)

KeyError: 22

Example 3: This method can be easily implemented into program to perform some business logic.

set = {'i','n','d','i','a','i','s','a','c','o','u','n','t','r','y'}

set2 = {'i','n','d','i','a','i','s','a','c','o','u','n','t','r','y'}

list = ['a','e','i','o','u']

print(set)

for el in set:

if el not in list:

set2.remove(el) # Removing elements which are not in list

print(set2)

Output:

{'a', 'c', 'i', 't', 'n', 'u', 'y', 's', 'd', 'o', 'r'}

{'a', 'i', 'u', 'o'}

Example-2 **Using remove() function**

months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(months)

print("\nRemoving some months from the set...");

months.remove("January");

months.remove("May");

print("\nPrinting the modified set...");

print(months)

Output:

printing the original set ...

{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...

{'February', 'June', 'April', 'March'}

Example-3 **Error due to remove method**

Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nRemoving items through discard() method...");

Months.discard("Feb"); #will not give an error although the key feb is not available in the set

print("\nprinting the modified set...")

print(Months)

print("\nRemoving items through remove() method...");

Months.remove("Jan") #will give an error as the key jan is not available in the set.

print("\nPrinting the modified set...")

print(Months)

Output:

printing the original set ...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...

Traceback (most recent call last):

File "set.py", line 9, in

Months.remove("Jan")

KeyError: 'Jan'

**pop() method:**

We can also use the pop() method to remove the item. Generally, **the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set**. It does not take any argument but **returns the popped element**. **It raises an error if the element is not present in the set**.

Syntax: pop()

Parameters: No parameter.

Return: It returns deleted element or throws an error if the set is empty.

Example 1: A simple example to use pop() method which removes an element and modifies the set.

set = {1,2,3,4,5}

print(set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

Output:

{1, 2, 3, 4, 5}

Element popped: 1

Remaining elements: {2, 3, 4, 5}

Element popped: 2

Remaining elements: {3, 4, 5}

Element popped: 3

Remaining elements: {4, 5}

Element popped: 4

Remaining elements: {5}

Example 2: If the set is empty, it throws an error KeyError to the caller function. See the example.

set = {1,2}

print(set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

Output:

{1, 2}

Element popped: 1

Remaining elements: {2}

Element popped: 2

Remaining elements: set()

Traceback (most recent call last):

File "main.py", line 13, in

el = set.pop()

KeyError: 'pop from an empty set'

Example 3: This example contains adding and poping elements in sequence to describe the functioning.

set = {1,2}

print(set)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

set.add(4)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

set.add(5)

el = set.pop()

print("Element popped:",el)

print("Remaining elements: ",set)

Output:

{1, 2}

Element popped: 1

Remaining elements: {2}

Element popped: 2

Remaining elements: {4}

Element popped: 4

Remaining elements: {5}

Example:

Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nRemoving some months from the set...")

del1=Months.pop()

print(del1,”deleted”)

del2=Months.pop()

print(del2,”deleted”)

print("\nPrinting the modified set...")

print(Months)

Output:

printing the original set ...

{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

January deleted

June deleted

Printing the modified set...

{'May', 'April', 'February', 'March'}

In the above code, the last element of the Month set is March but the pop() method removed the June and January because the set is unordered and the pop() method could not determine the last element of the set.

**clear() method:**

Python provides the clear() method to remove all the items from the set.

Example:

Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nRemoving all the items from the set...");

Months.clear()

print("\nPrinting the modified set...")

print(Months)

Output:

printing the original set ...

{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...

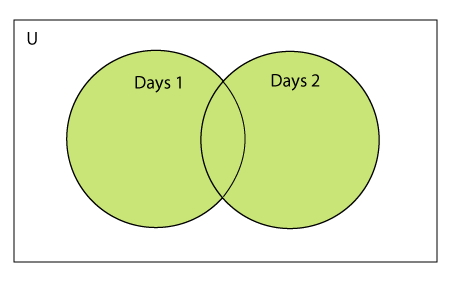
set()

**Python Set Operations**

Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

**Union of two Sets**

The union of two sets is calculated by using the **pipe (|) operator or union() method**. The union of the two sets contains all the items that are present in both the sets.



Example 1: **using union | operator**

Days1 = {"Monday","Tuesday","Wednesday","Thursday", "Sunday"}

Days2 = {"Friday","Saturday","Sunday"}

print(Days1|Days2)

Output:

{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}

Python also provides the union() method which can also be used to calculate the union of two sets. Consider the following example.

Example 2: **using union() method**

Days1 = {"Monday","Tuesday","Wednesday","Thursday"}

Days2 = {"Friday","Saturday","Sunday"}

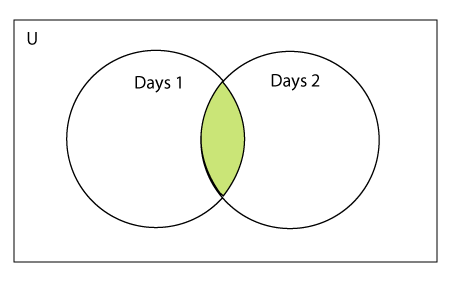
print(Days1.union(Days2))

Output:

{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}

**Intersection of two sets**

The intersection of two sets can be performed by the **and & operator or the intersection() function**. The intersection of the two sets is given as the set of the elements that common in both sets.



Example 1: **Using & operator**

Days1 = {"Monday","Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday","Tuesday","Sunday", "Friday"}

print(Days1&Days2)

Output:

{'Monday', 'Tuesday'}

Example 2: **Using intersection() method**

set1 = {"Devansh","John", "David", "Martin"}

set2 = {"Steve", "Milan", "David", "Martin"}

print(set1.intersection(set2))

Output:

{'Martin', 'David'}

Example 3:

set1 = {1,2,3,4,5,6,7}

set2 = {1,2,20,32,5,9}

set3 = set1.intersection(set2)

print(set3)

Output:

{1,2,5}

**The intersection\_update() method**

The intersection\_update() method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

**The intersection\_update() method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.**

Example:

a = {"Devansh", "bob", "castle"}

b = {"castle", "dude", "emyway"}

c = {"fuson", "gaurav", "castle"}

a.intersection\_update(b, c)

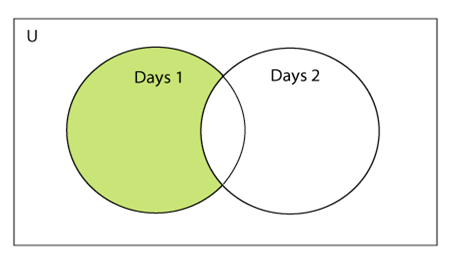
print(a)

Output:

{'castle'}

**Difference between the two sets**

The difference of two sets can be calculated by using the **subtraction (-) operator or difference() method**. Suppose there are two sets A and B, and the difference is A-B that denotes the resulting set will be obtained that element of A, which is not present in the set B.



Example 1 : **Using subtraction ( - ) operator**

Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday", "Tuesday", "Sunday"}

print(Days1-Days2)

Output:

{'Thursday', 'Wednesday'}

Example 2 : **Using difference() method**

Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday", "Tuesday", "Sunday"}

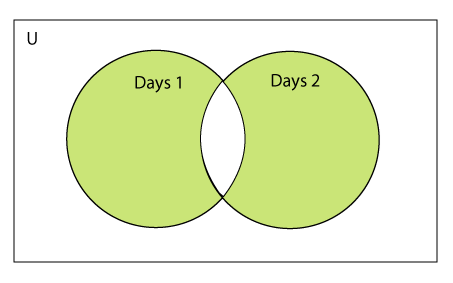
print(Days1.difference(Days2))

Output:

{'Thursday', 'Wednesday'}

**Symmetric Difference of two sets**

The symmetric difference of two sets is calculated **by ^ operator or symmetric\_difference() method**. Symmetric difference of sets, it removes that element which is present in both sets.



Example - 1: **Using ^ operator**

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

b = {1,2,9,8,10}

c = a^b

print(c)

Output:

{3, 4, 5, 6, 8, 9, 10}

Example - 2: **Using symmetric\_difference() method**

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

b = {1,2,9,8,10}

c = a.symmetric\_difference(b)

print(c)

Output:

{3, 4, 5, 6, 8, 9, 10}

**Set comparisons**

**Python allows us to use the comparison operators i.e., <, >, <=, >= , == with the sets by using which we can check whether a set is a subset, superset, or equivalent to other set**. The boolean true or false is returned depending upon the items present inside the sets.

Example:

Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday", "Tuesday"}

Days3 = {"Monday", "Tuesday", "Friday"}

#Days1 is the superset of Days2 hence it will print true.

print (Days1>Days2)

#prints false since Days1 is not the subset of Days2

print (Days1<Days2)

#prints false since Days2 and Days3 are not equivalent

print (Days2 == Days3)

Output:

True

False

False

**FrozenSets**

The frozen sets are the **immutable form of the normal sets**, i.e., **the items of the frozen set cannot be changed** and therefore it can be used as a key in the dictionary.

The elements of the frozen set cannot be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Example:

Frozenset = frozenset([1,2,3,4,5])

print(type(Frozenset))

print("\nprinting the content of frozen set...")

for i in Frozenset:

print(i);

Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation

Output:

<class 'frozenset'>

printing the content of frozen set...

1

2

3

4

5

Traceback (most recent call last):

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

Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation

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

**Frozenset for the dictionary**

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

Example:

Dictionary = {"Name":"John", "Country":"USA", "ID":101}

print(type(Dictionary))

Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary

print(type(Frozenset))

for i in Frozenset:

print(i)

Output:

<class 'dict'>

<class 'frozenset'>

Name

Country

ID

**Python Built-in set methods**

Python contains the following methods to be used with the sets.

|  |  |  |
| --- | --- | --- |
| SN | Method | Description |
| 1 | add(item) | It adds an item to the set. It has no effect if the item is already present in the set. |
| 2 | clear() | It deletes all the items from the set. |
| 3 | copy() | It returns a shallow copy of the set. |
| 4 | difference\_update(....) | It modifies this set by removing all the items that are also present in the specified sets. |
| 5 | discard(item) | It removes the specified item from the set. |
| 6 | intersection() | It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified). |
| 7 | intersection\_update(....) | It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified). |
| 8 | Isdisjoint(....) | Return True if two sets have a null intersection. |
| 9 | Issubset(....) | Report whether another set contains this set. |
| 10 | Issuperset(....) | Report whether this set contains another set. |
| 11 | pop() | Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty. |
| 12 | remove(item) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 13 | symmetric\_difference(....) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 14 | symmetric\_difference\_update(....) | Update a set with the symmetric difference of itself and another. |
| 15 | union(....) | Return the union of sets as a new set. (i.e. all elements that are in either set.) |
| 16 | update() | Update a set with the union of itself and others. |

**Set Programming Example**

Example - 1: Write a program to remove the given number from the set.

my\_set = {1,2,3,4,5,6,12,24}

n = int(input("Enter the number you want to remove"))

my\_set.discard(n)

print("After Removing:",my\_set)

Output:

Enter the number you want to remove:12

After Removing: {1, 2, 3, 4, 5, 6, 24}

Example - 2: Write a program to add multiple elements to the set.

set1 = set([1,2,4,"John","CS"])

set1.update(["Apple","Mango","Grapes"])

print(set1)

Output:

{1, 2, 4, 'Apple', 'John', 'CS', 'Mango', 'Grapes'}

Example - 5: Write the program to create new frozenset after applying intersection on two frozensets.

set1 = frozenset({23,44,56,67,90,45,"Python"})

set2 = frozenset({13,23,56,76,"Sachin"} )

set3 = set1.intersection(set2)

print(set3)

Output:

frozenset({56, 23})

Example - 6: Write the program to find the issuperset, issubset and superset.

set1 = set(["Peter","James","Camroon","Ricky","Donald"])

set2 = set(["Camroon","Washington","Peter"])

set3 = set(["Peter"])

issubset = set1 >= set2

print(issubset)

issuperset = set1 <= set2

print(issuperset)

issubset = set3 <= set2

print(issubset)

issuperset = set2 >= set3

print(issuperset)

Output:

False

False

True

True

**Python Dictionary**

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. **It is the mutable data-structure.** **The dictionary is defined into element Keys and values.**

**Keys must be a single 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 any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

**Creating the dictionary**

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

Syntax:

Dict = {"Name": "Tom", "Age": 22}

In the above dictionary Dict, The keys Name and Age are the string that is an immutable object.

Example:

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)

Output

<class 'dict'>

Printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

**Python provides the built-in function dict() method which is also used to create dictionary**. **The empty curly braces {} is used to create empty dictionary.**

Example:

# Creating an empty Dictionary

Dict = {}

print("Empty Dictionary: ")

print(Dict)

# Creating a Dictionary with dict() method

Dict = dict({1: 'Python', 2: 'Java', 3:'C++'})

print("\nCreate Dictionary by using dict(): ")

print(Dict)

# Creating a Dictionary with each item as a Pair

Dict = dict([(1, 'Devansh'), (2, 'Sharma')])

print("\nDictionary with each item as a pair: ")

print(Dict)

Output:

Empty Dictionary:

{}

Create Dictionary by using dict():

{1: 'Python', 2: 'Java', 3: 'C++'}

Dictionary with each item as a pair:

{1: 'Devansh', 2: 'Sharma'}

**Accessing the dictionary values**

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

Syntax: Dictionaryname[Key]

Example:

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print("Name : %s" %Employee["Name"])

print("Age : %d" %Employee["Age"])

print("Salary : %d" %Employee["salary"])

print("Company : %s" %Employee["Company"])

Output:

<class 'dict'>

printing Employee data ....

Name : John

Age : 29

Salary : 25000

Company : GOOGLE

**Python provides us with an alternative to use the get() method to access the dictionary values.** It would give the same result as given by the indexing.

**Adding dictionary values**

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key Dict[key] = value. **The update() method is also used to update an existing value**.

Note: **If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.**

Example - 1:

# Creating an empty Dictionary

Dict = {}

print("Empty Dictionary: ")

print(Dict)

# Adding elements to dictionary one at a time

Dict[0] = 'Peter'

Dict[2] = 'Joseph'

Dict[3] = 'Ricky'

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

print(Dict)

# Adding set of values with a single Key

Dict['Emp\_ages'] = 20, 33, 24

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

print(Dict)

# Updating existing Key's Value

Dict[3] = 'Python'

print("\nUpdated key value: ")

print(Dict)

Output:

Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'Python', 'Emp\_ages': (20, 33, 24)}

Example - 2: Accept value for key from user and update the existing value.

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)

print("Enter the details of the new employee....");

**Employee["Name"] = input("Name: ");**

Employee["Age"] = int(input("Age: "));

Employee["salary"] = int(input("Salary: "));

Employee["Company"] = input("Company:");

print("printing the new data");

print(Employee)

Output:

Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'Python', 'Emp\_ages': (20, 33, 24)}

**Deleting elements using del keyword**

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

Example:

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)

print("Deleting some of the employee data")

**del Employee["Name"]**

del Employee["Company"]

print("printing the modified information ")

print(Employee)

print("Deleting the dictionary: Employee");

**del Employee**

print("Lets try to print it again ");

print(Employee)

Output:

<class 'dict'>

printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Deleting some of the employee data

printing the modified information

{'Age': 29, 'salary': 25000}

Deleting the dictionary: Employee

Lets try to print it again

NameError: name 'Employee' is not defined

**Using pop() method**

The pop() method accepts the key as an argument and remove the associated value.

Example:

Dict = {1: 'Python', 2: 'Peter', 3: 'Thomas'}

pop\_ele = Dict.pop(3) #parameter is key

print(Dict)

Output:

{1: 'Python', 2: 'Peter'}

Python also provides a built-in methods **popitem() and clear()** method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

**Iterating Dictionary**

A dictionary can be iterated using for loop as given below.

Example 1: for loop to print all the keys of a dictionary

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee:

print(x)

Output:

Name

Age

salary

Company

Example 2: for loop to print all the values of the dictionary

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee:

print(Employee[x])

Output:

John

29

25000

GOOGLE

Example 3: for loop to print all the keys of a dictionary **using keys() function**.

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee.keys():

print(x)

Output:

Name

Age

salary

Company

Example – 4: for loop to print the values of the dictionary by using **values()** method.

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee.values():

print(x)

Output:

John

29

25000

GOOGLE

Example 5: for loop to print the items of the dictionary by using **items()** method.

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee.items():

print(x)

Output:

('Name', 'John')

('Age', 29)

('salary', 25000)

('Company', 'GOOGLE')

Example 6: for loop to print the items of the dictionary by using **items()** method. Store key and value separately.

Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}

for x,y in Employee.items():

print(x,y)

Output:

Name John

Age 29

Salary 25000

Company GOOGLE

**Properties of Dictionary keys**

1. In the dictionary, **we cannot store multiple values for the same keys**. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key. Value can be set of data.

Example:

Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}

for x,y in Employee.items():

print(x,y)

Output:

Name John

Age 29

Salary 25000

Company GOOGLE

2. In python, **the key cannot be any mutable object.** We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Example:

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}

for x,y in Employee.items():

print(x,y)

Output:

Traceback (most recent call last):

File "dictionary.py", line 1, in

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}

TypeError: unhashable type: 'list'

**Built-in Dictionary functions**

The built-in python dictionary methods along with the description are given below.

|  |  |  |
| --- | --- | --- |
| SN | Function | Description |
| 1 | len(dict) | It is used to calculate the length of the dictionary. |
| 2 | str(dict) | It converts the dictionary into the printable string representation. |

**Built-in Dictionary methods**

The built-in python dictionary methods along with the description are given below.

|  |  |  |
| --- | --- | --- |
| SN | Method | Description |
| 1 | dict.clear() | It is used to delete all the items of the dictionary.  Employee.clear() |
| 2 | dict.copy() | It returns a shallow copy of the dictionary.  Employee1 = Employee.copy() |
| 3 | dict.fromkeys(iterable, value = None, /) | Create a new dictionary from the iterable with the values equal to value.  x = ('key1', 'key2', 'key3')  y = 0  thisdict = dict.fromkeys(x, y)  print(thisdict)  {'key1': 0, 'key2': 0, 'key3': 0} |
| 4 | dict.get(key) | It is used to get the value specified for the passed key. |
| 5 | dict.items() | It returns all the key-value pairs as a tuple. |
| 6 | dict.keys() | It returns all the keys of the dictionary. |
| 7 | dict.setdefault(key,default= "None") | Python setdefault() method is used to set default value to the key. It returns value, if the key is present. Otherwise it insert key with the default value. Default value for the key is None. |
| 8 | dict.update(dict2) | It updates the dictionary by adding the key-value pair of dict2 to this dictionary. |
| 9 | dict.values() | It returns all the values of the dictionary. |
| 10 | popItem() |  |
| 11 | pop() |  |
| 12 | count() |  |
| 13 | index() |  |

**Python Dictionary items() Method**

Python item() method returns a new view of the dictionary. This view is collection of key value tuples. This method does not take any parameter and returns empty view if the dictionary is empty.

Syntax: items()

Parameters: No parameter

Return: It returns a dictionary's view.

Example 1:This is a simple example which returns all the items present in the dictionary.

student = {'name':'rohan', 'course':'B.Tech', 'email':'rohan@abc.com'}

items = student.items()

print(items)

Output:

dict\_items([('name', 'rohan'), ('course', 'B.Tech'), ('email', 'rohan@abc.com')])

Example 2: If the dictionary is already empty, this method does not raise any error.

student = {} # dictionary is empty

items = student.items()

print(items)

Output:

dict\_items([])

Example 3: Apart from items() method, we can also use other customized approaches to fetch dictionary elements.

student = {'name':'rohan', 'course':'B.Tech', 'email':'rohan@abc.com'}

# Iterating using key and value

for st in student:

print("(",st, ":", student[st], end="), ")

items = student.items()

print("\n", items)

Output:

( name : rohan), ( course : B.Tech), ( email : rohan@abc.com),

dict\_items([('name', 'rohan'), ('course', 'B.Tech'), ('email', 'rohan@abc.com')])

**Python Dictionary keys() Method**

Python keys() method is used to fetch all the keys from the dictionary. It returns a list of keys and an empty list if the dictionary is empty. This method does not take any parameter.

Syntax: keys()

Parameters: No parameter

Return: It returns a list of keys. None if the dictionary is empty.

Example 1: Let's first see a simple example to fetch keys from the dictionary.

product = {'name':'laptop','brand':'hp','price':80000}

p = product.keys()

print(p)

Output:

dict\_keys(['name', 'brand', 'price'])

Example 2: Python dictionary keys() Method

product = {'name':'laptop','brand':'hp','price':80000}

for p in product.keys():

if p == 'price' and product[p] > 50000:

print("product price is too high",)

Output:

product price is too high

Example 3: Here, we are using it into a program to check our inventory status.

inventory = {'apples': 25, 'bananas': 220, 'oranges': 525, 'pears': 217}

for akey in inventory.keys():

if akey == 'bananas' and inventory[akey] > 200:

print("We have sufficient inventory for the ", akey)

Output:

We have sufficient inventory for the bananas

**Python Dictionary values() Method**

Python values() method is used to collect all the values from a dictionary. It does not take any parameter and returns a dictionary of values. It returns an empty dictionary if the dictionary has no value.

Signature: values()

Parameters: No Parameter

Return: It returns a dictionary of values.

Example 1: It is a simple example which returns all the values from the dictionary. An example is given below.

inventory = {'Fan': 200, 'Bulb':150, 'Led':1000}

stock = einventory.values()

print("Stock available",einventory)

Output:

Stock available {'Fan': 200, 'Bulb': 150, 'Led': 1000}

Example 2: If the dictionary is already empty, this method also returns an empty dictionary except any error or exception.

einventory = {}

stock = einventory.values()

print("Stock available",einventory)

Output:

Stock available {}

Example 3: This example uses various methods to get information about dictionary such as length, keys etc.

einventory = {'Fan': 200, 'Bulb':150, 'Led':1000}

length = len(einventory)

print("Total number of values:",length)

keys = einventory.keys()

print("All the Keys:",keys)

item = einventory.items()

print("Items:",einventory)

p = einventory.popitem()

print("Deleted items:",p)

stock = einventory.values()

print("Stock available",einventory)

Output:

Total number of values: 3

All the Keys: dict\_keys(['Fan', 'Bulb', 'Led'])

Items: {'Fan': 200, 'Bulb': 150, 'Led': 1000}

Deleted items: ('Led', 1000)

Stock available {'Fan': 200, 'Bulb': 150}

**Python Dictionary setdefault() Method**

Python setdefault() method is used to set default value to the key. It returns value, if the key is present. Otherwise it insert key with the default value. Default value for the key is None.

Signature: setdefault(key[, default])

Parameters: key: key to be searched. default: This value to be returned, if the key is not found.

Return: It returns a value, if the key is present. Otherwise None or default value.

Example 1: A simple example, if key is present, it returns associated value.

coursefee = {'B,Tech': 400000, 'BA':2500, 'B.COM':50000}

p = coursefee.setdefault('BA') # Returns it's value

print("default",p)

print(coursefee)

Output:

default 2500

{'B,Tech': 400000, 'BA': 2500, 'B.COM': 50000}

Example 2: If neither key nor default value is present, it returns None.

coursefee = {'B,Tech': 400000, 'BA':2500, 'B.COM':50000}

p = coursefee.setdefault('BCA') # Returns it's value

print("default",p)

print(coursefee)

Output:

default None

{'B,Tech': 400000, 'BA': 2500, 'B.COM': 50000, 'BCA': None}

Example 3: If key is not present but default value is set, it returns default value. See an example.

coursefee = {'B,Tech': 400000, 'BA':2500, 'B.COM':50000}

p = coursefee.setdefault('BCA',100000) # Returns it's value

print("default",p)

print(coursefee)

Output:

default 100000

{'B,Tech': 400000, 'BA': 2500, 'B.COM': 50000, 'BCA': 100000}

**Python Dictionary update() Method**

Python update() method updates the dictionary with the key and value pairs. It inserts key/value if it is not present. It updates key/value if it is already present in the dictionary.

It also allows an iterable of key/value pairs to update the dictionary. like: update(a=10,b=20) etc.

Signature: update([other])

Parameters: other: It is a list of key/value pairs.

Return: It returns None.

Example 1: It is a simple example to update the dictionary by passing key/value pair. This method updates the dictionary.

einventory = {'Fan': 200, 'Bulb':150, 'Led':1000}

print("Inventory:",einventory)

einventory.update({'cooler':50})

print("Updated inventory:",einventory)

Output:

Inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000}

Updated inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000, 'cooler': 50}

Example 2: If element (key/value) pair is already presents in the dictionary, it will overwrite it.

einventory = {'Fan': 200, 'Bulb':150, 'Led':1000,'cooler':50}

print("Inventory:",einventory)

einventory.update({'cooler':50})

print("Updated inventory:",einventory)

einventory.update({'cooler':150})

print("Updated inventory:",einventory)

Output:

Inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000, 'cooler': 50}

Updated inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000, 'cooler': 50}

Updated inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000, 'cooler': 150}

Example 3: The update() method also allows iterable key/value pairs as parameter. See, the example below two values are passed to the dictionary and it is updated.

einventory = {'Fan': 200, 'Bulb':150, 'Led':1000}

print("Inventory:",einventory)

einventory.update(cooler=50,switches=1000)

print("Updated inventory:",einventory)

Output:

Inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000}

Updated inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000, 'cooler': 50, 'switches': 1000}

**Python Dictionary popitem() Method**

Python popitem() method removes an element from the dictionary. It removes arbitrary element and return its value. If the dictionary is empty, it returns an error KeyError.

Signature: popitem()

Parameters: No parameter

Return: It returns the popped element.

Example 1: Let's first see a simple example to remove an element using popitem() method.

inventory = {'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}

print(inventory)

p = inventory.popitem()

print("Removed",p)

print(inventory)

Output:

{'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}

Removed ('tshirts', 217)

{'shirts': 25, 'paints': 220, 'shocks': 525}

Example 2: If the dictionary is empty, it returns an error KeyError.

inventory = {}

print(inventory)

p = inventory.popitem()

print("Removed",p)

print(inventory)

Output:

KeyError: 'popitem(): dictionary is empty'

Example 3: In this example, we are removing and updating the dictionary to understand the functioning of this method.

inventory = {'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}

print(inventory)

p = inventory.popitem()

print("Removed",p)

print(inventory)

inventory.update({'pajama':117})

print(inventory)

Output:

{'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}

Removed ('tshirts', 217)

{'shirts': 25, 'paints': 220, 'shocks': 525}

{'shirts': 25, 'paints': 220, 'shocks': 525, 'pajama': 117}

**Python Dictionary pop() Method**

Python pop() method removes an element from the dictionary. It removes the element which is associated to the specified key.

If specified key is present in the dictionary, it remove and return its value.

If the specified key is not present, it throws an error KeyError.

Syntax: pop(key[, default])

Parameters: key: A key to delete value associated to it.

defaults: If key is not present, defaults value is returned.

Return: It removes and returns value associated to the specified key.

Example 1:A simple example to pop an element from the dictionary. It returns popped value.

inventory = {'shirts': 25, 'paints': 220, 'shock': 525, 'tshirts': 217}

element = inventory.pop('shirts')

print(element)

Output:

25

Example 2: If the key is not present, it returns an error KeyError.

inventory = {'shirts': 25, 'paints': 220, 'shock': 525, 'tshirts': 217}

element = inventory.pop('shoes')

print(element)

Output:

KeyError: 'shoes'

Example 3: If key is not present, we can set default value to avoid the error KeyError.

inventory = {'shirts': 25, 'paints': 220, 'shock': 525, 'tshirts': 217}

element = inventory.pop('shoes',100)

print(element)

Output:

100

**Python Function**

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

Python provide us various inbuilt functions like range() or print(). Although, the user can create its functions, which can be called user-defined functions.

**There are mainly two types of functions.**

**User-define functions** - The user-defined functions are those define by the user to perform the specific task.

**Built-in functions** - The built-in functions are those functions that are pre-defined in Python.

**Advantage of Functions in Python**

There are the following advantages of Python functions.

* Using functions, we can avoid rewriting the same logic/code again and again in a program.
* We can call Python functions multiple times in a program and anywhere in a program.
* We can track a large Python program easily when it is divided into multiple functions.
* Reusability is the main achievement of Python functions.

**Creating a Function**

Python provides the **def** keyword to define the function.

Syntax:

def function\_name(parameters):

function\_block

return expression

* The def keyword, along with the function name is used to define the function.
* The identifier rule must follow the function name.
* A function accepts the parameter (argument), and they can be optional.
* The function block is started with the colon (:), and block statements must be at the same indentation.
* The return statement is used to return the value. A function can have only one return

**Function Calling**

In Python, **after the function is created, we can call it from another function**. **A function must be defined before the function call**; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

Example:

#function definition

def hello\_world():

print("hello world")

# function calling

hello\_world()

Output:

hello world

**The return statement**

The return statement is used at the end of the function and returns the result of the function. **It terminates the function execution and transfers the result where the function is called**. The return statement cannot be used outside of the function.

Syntax

return [expression\_list]

It can contain the expression which gets evaluated and value is returned to the caller function. **If the return statement has no expression or does not exist itself in the function then it returns the None object.**

Example 1

def sum():

a = 10

b = 20

c = a+b

return c

# calling sum() function in print statement

print("The sum is:",sum())

Output:

The sum is: 30

In the above code, we have defined the function named sum, and it has a statement c = a+b, which computes the given values, and the result is returned by the return statement to the caller function.

Example 2 Creating function without return statement

def sum():

a = 10

b = 20

c = a+b

# calling sum() function in print statement

print(sum())

Output:

None

In the above code, we have defined the same function without the return statement as we can see that the sum() function returned the None object to the caller function.

**Arguments in function**

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

Example 1

def func (name):

print("Hi ",name)

func("Devansh")

Output:

Hi Devansh

Example 2- Python function to calculate the sum of two variables

def sum (a,b):

return a+b;

a = int(input("Enter a: "))

b = int(input("Enter b: "))

print("Sum = ",sum(a,b))

Output:

Enter a: 10

Enter b: 20

Sum = 30

**Call by reference in Python**

In Python, **call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.**

Example 1 Passing mutable Object (List)

def change\_list(list1):

list1.append(20)

list1.append(30)

print("list inside function = ",list1)

list1 = [10,30,40,50]

change\_list(list1)

print("list outside function = ",list1)

Output:

list inside function = [10, 30, 40, 50, 20, 30]

list outside function = [10, 30, 40, 50, 20, 30]

Example 2 Passing immutable Object (String)

def change\_string (str):

str = str + " Hows you "

print("printing the string inside function :",str)

string1 = "Hi I am there"

#calling the function

change\_string(string1)

print("printing the string outside function :",string1)

Output:

printing the string inside function : Hi I am there Hows you

printing the string outside function : Hi I am there

**Types of arguments**

There may be several types of arguments which can be passed at the time of function call.

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

**Required Arguments**

Till now, we have learned about function calling in Python. However, we can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition**. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.**

Example 1

def func(name):

message = "Hi "+name

return message

name = input("Enter the name:")

print(func(name))

Output:

Enter the name: John

Hi John

Example 2- the function simple\_interest accepts three arguments and returns the simple interest accordingly

def simple\_interest(p,t,r):

return (p\*t\*r)/100

p = float(input("Enter the principle amount? "))

r = float(input("Enter the rate of interest? "))

t = float(input("Enter the time in years? "))

print("Simple Interest: ",simple\_interest(p,r,t))

Output:

Enter the principle amount: 5000

Enter the rate of interest: 5

Enter the time in years: 3

Simple Interest: 750.0

Example 3- the function calculate returns the sum of two arguments a and b

def calculate(a,b):

return a+b

calculate(10) # this causes an error as we are missing a required arguments b.

Output:

TypeError: calculate() missing 1 required positional argument: 'b'

**Default Arguments**

Python allows us to initialize the arguments at the function definition. **If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.**

Example 1

def printme(name,age=22):

print("My name is",name,"and age is",age)

printme(name = "john")

Output:

My name is John and age is 22

Example 2

def printme(name,age=22):

print("My name is",name,"and age is",age)

printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function

printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age

Output:

My name is john and age is 22

My name is David and age is 10

**Variable-length Arguments (\*args)**

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

However, at the function definition, we define the variable-length argument using the \*args (star) as \*<variable - name >.

Example

def printme(\*names):

print("type of passed argument is ",type(names))

print("printing the passed arguments...")

for name in names:

print(name)

printme("john","David","smith","nick")

Output:

type of passed argument is <class 'tuple'>

printing the passed arguments...

john

David

smith

nick

In the above code, we passed \*names as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated \*arg names using for loop.

**Keyword arguments(\*\*kwargs)**

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Example 1

#function func is called with the name and message as the keyword arguments

def func(name,message):

print("printing the message with",name,"and ",message)

#name and message is copied with the values John and hello respectively

func(name = "John",message="hello")

Output:

printing the message with John and hello

Example 2 providing the values in different order at the calling

#The function simple\_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case

def simple\_interest(p,t,r):

return (p\*t\*r)/100

print("Simple Interest: ",simple\_interest(t=10,r=10,p=1900))

Output:

Simple Interest: 1900.0

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

Example 3

#The function simple\_interest(p, t, r) is called with the keyword arguments.

def simple\_interest(p,t,r):

return (p\*t\*r)/100

# doesn't find the exact match of the name of the arguments (keywords)

print("Simple Interest: ",simple\_interest(time=10,rate=10,principle=1900))

Output:

TypeError: simple\_interest() got an unexpected keyword argument 'time'

The Python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., **once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.**

Example 4

def func(name1,message,name2):

print("printing the message with",name1,",",message,",and",name2)

#the first argument is not the keyword argument

func("John",message="hello",name2="David")

Output:

printing the message with John , hello ,and David

Example 5

def func(name1,message,name2):

print("printing the message with",name1,",",message,",and",name2)

func("John",message="hello","David")

Output:

SyntaxError: positional argument follows keyword argument

Python provides the facility to pass the multiple keyword arguments which can be represented as \*\*kwargs. It is similar as the \*args but it stores the argument in the dictionary format.

**This type of arguments is useful when we do not know the number of arguments in advance.**

Example 6: Many arguments using Keyword argument

def food(\*\*kwargs):

print(kwargs)

food(a="Apple")

food(fruits="Orange", Vagitables="Carrot")

Output:

{'a': 'Apple'}

{'fruits': 'Orange', 'Vagitables': 'Carrot'}

**Scope of variables**

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

**In python, the variables are defined with the two types of scopes.**

* Global variables
* Local variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.

**Local Variable**

Local variables are the variables that declared inside the function and have scope within the function. Let's understand the following example.

Example -

def add():

# Defining local variables. They has scope only within a function

a = 20

b = 30

c = a + b

print("The sum is:", c)

# Calling a function

add()

Output:

The sum is: 50

Explanation:

In the above code, we declared a function named add() and assigned a few variables within the function. These variables will be referred to as the local variables which have scope only inside the function. If we try to use them outside the function, we get a following error.

add()

# Accessing local variable outside the function

print(a)

Output:

The sum is: 50

print(a)

NameError: name 'a' is not defined

We tried to use local variable outside their scope; it threw the NameError.

**Global Variables**

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.

A variable declared outside the function is the global variable by default. Python provides the global keyword to use global variable inside the function. If we don't use the global keyword, the function treats it as a local variable.

Example -

# Declare a variable and initialize it

x = 101

# Global variable in function

def mainFunction():

# printing a global variable

global x

print(x)

# modifying a global variable

x = 'Welcome To Python'

print(x)

mainFunction()

print(x)

Output:

101

Welcome To Python

Welcome To Python

Explanation:

In the above code, we declare a global variable x and assign a value to it. Next, we defined a function and accessed the declared variable using the global keyword inside the function. Now we can modify its value. Then, we assigned a new string value to the variable x.

Now, we called the function and proceeded to print x. It printed the as newly assigned value of x.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

x = "awesome"

def myfunc():

print("Python is " + x)

myfunc()

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example-Create a variable inside a function, with the same name as the global variable

x = "awesome"

def myfunc():

x = "fantastic"

print("Python is " + x)

myfunc()

print("Python is " + x)

The global Keyword

To create a global variable inside a function, you can use the global keyword.

Example

If you use the global keyword, the variable belongs to the global scope:

def myfunc():

global x

x = "fantastic"

myfunc()

print("Python is " + x)

Also, use the global keyword if you want to change a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

x = "awesome"

def myfunc():

global x

x = "fantastic"

myfunc()

print("Python is " + x)

Example 1 Local Variable

def print\_message():

message = "hello !! I am going to print a message." # the variable message is local to the function itself

print(message)

print\_message()

print(message) # this will cause an error since a local variable cannot be accessible here.

Output:

hello !! I am going to print a message.

File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in

print(message)

NameError: name 'message' is not defined

Example 2 Local Variable

def calculate(\*args):

sum=0

for arg in args:

sum = sum +arg

print("The sum is",sum)

sum=0

calculate(10,20,30) #60 will be printed as the sum

print("Value of sum outside the function:",sum) # 0 will be printed Output:

Output:

The sum is 60

Value of sum outside the function: 0

**Python Built-in Functions**

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

**Python abs() Function**

The python abs() function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

Python abs() Function Example

# integer number

integer = -20

print('Absolute value of -20 is:', abs(integer))

# floating number

floating = -20.83

print('Absolute value of -20.83 is:', abs(floating))

Output:

Absolute value of -20 is: 20

Absolute value of -20.83 is: 20.83

**Python all() Function**

The python all() function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

Python all() Function Example

# all values true

k = [1, 3, 4, 6]

print(all(k))

# all values false

k = [0, False]

print(all(k))

# one false value

k = [1, 3, 7, 0]

print(all(k))

# one true value

k = [0, False, 5]

print(all(k))

# empty iterable

k = []

print(all(k))

Output:

True

False

False

False

True

**Python any() Function**

The python any() function returns true if any item in an iterable is true. Otherwise, it returns False.

Python any() Function Example

l = [4, 3, 2, 0]

print(any(l))

l = [0, False]

print(any(l))

l = [0, False, 5]

print(any(l))

l = []

print(any(l))

Output:

True

False

True

False

**Python bin() Function**

The python bin() function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Python bin() Function Example

x = 10

y = bin(x)

print (y)

Output:

0b1010

In same way we can use oct() for octal and hex() for hexadecimal.

**Python bool()**

The python bool() converts a value to boolean(True or False) using the standard truth testing procedure.

Python bool() Example

test1 = []

print(test1,'is',bool(test1))

test1 = [0]

print(test1,'is',bool(test1))

test1 = 0.0

print(test1,'is',bool(test1))

test1 = None

print(test1,'is',bool(test1))

test1 = True

print(test1,'is',bool(test1))

test1 = 'Easy string'

print(test1,'is',bool(test1))

Output:

[] is False

[0] is True

0.0 is False

None is False

True is True

Easy string is True

**Python callable() Function**

A python callable() function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

Python callable() Function Example

x = 8

print(callable(x))

Output:

False

**Python compile() Function**

The python compile() function takes source code as input and returns a code object which can later be executed by exec() function.

Python compile() Function Example

# compile string source to code

code\_str = 'x=5\ny=10\nprint("sum =",x+y)'

code = compile(code\_str, 'sum.py', 'exec')

print(type(code))

exec(code)

Output:

<class 'code'>

sum = 15

**Python exec() Function**

The python exec() function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

Python exec() Function Example

x = 8

exec('print(x==8)')

exec('print(x+4)')

Output:

True

12

**Python sum() Function**

As the name says, python sum() function is used to get the sum of numbers of an iterable, i.e., list.

Python sum() Function Example

s = sum([1, 2,4 ])

print(s)

s = sum([1, 2, 4], 10)

print(s)

Output:

7

17

**Python eval() Function**

The python eval() function parses the expression passed to it and runs python expression(code) within the program.

Python eval() Function Example

x = 8

print(eval('x + 1'))

Output:

9

**Python float()**

The python float() function returns a floating-point number from a number or string.

Python float() Example

# for integers

print(float(9))

# for floats

print(float(8.19))

# for string floats

print(float("-24.27"))

# for string floats with whitespaces

print(float(" -17.19\n"))

# string float error

print(float("xyz"))

Output:

9.0

8.19

-24.27

-17.19

ValueError: could not convert string to float: 'xyz'

**Python format() Function**

The python format() function returns a formatted representation of the given value.

Python format() Function Example

# d, f and b are a type

# integer

print(format(123, "d"))

# float arguments

print(format(123.4567898, "f"))

# binary format

print(format(12, "b"))

Output:

123

123.456790

1100

**Python frozenset()**

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

Python frozenset() Example

# tuple of letters

letters = ('m', 'r', 'o', 't', 's')

fSet = frozenset(letters)

print('Frozen set is:', fSet)

print('Empty frozen set is:', frozenset())

Output:

Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})

Empty frozen set is: frozenset()

Python **getattr()** Function

The python getattr() function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Python getattr() Function Example

class Details:

age = 22

name = "Phill"

details = Details()

print('The age is:', getattr(details, "age"))

print('The age is:', details.age)

Output:

The age is: 22

The age is: 22

**Python iter() Function**

The python iter() function is used to return an iterator object. It creates an object which can be iterated one element at a time.

Python iter() Function Example

# list of numbers

list = [1,2,3,4,5]

listIter = iter(list)

# prints '1'

print(next(listIter))

# prints '2'

print(next(listIter))

# prints '3'

print(next(listIter))

# prints '4'

print(next(listIter))

# prints '5'

print(next(listIter))

Output:

1

2

3

4

5

**Python len() Function**

The python len() function is used to return the length (the number of items) of an object.

Python len() Function Example

strA = 'Python'

print(len(strA))

Output:

6

**Python list()**

The python list() creates a list in python.

Python list() Example

# empty list

print(list())

# string

String = 'abcde'

print(list(String))

# tuple

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

print(list(Tuple))

# list

List = [1,2,3,4,5]

print(list(List))

Output:

[]

['a', 'b', 'c', 'd', 'e']

[1,2,3,4,5]

[1,2,3,4,5]

Python **open()** Function

The python open() function opens the file and returns a corresponding file object.

Python open() Function Example

# opens python.text file of the current directory

f = open("python.txt")

# specifying full path

f = open("C:/Python33/README.txt")

Output:

Since the mode is omitted, the file is opened in 'r' mode; opens for reading.

Python **chr()** Function

Python chr() function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

Python **delattr()** Function

Python delattr() function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Example:

class Student:

id = 101

name = "Pranshu"

email = "pranshu@abc.com"

# Declaring function

def getinfo(self):

print(self.id, self.name, self.email)

s = Student()

s.getinfo()

delattr(Student,'course') # Removing attribute which is not available

s.getinfo() # error: throws an error

Output:

101 Pranshu pranshu@abc.com

AttributeError: course

Python **divmod()** Function

Python divmod() function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

Example:

# Python divmod() function example

# Calling function

result = divmod(10,2)

# Displaying result

print(result)

Output:

(5, 0)

Python **dict()**

Python dict() function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

If no argument is passed, it creates an empty dictionary.

If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

Example

# Calling function

result = dict() # returns an empty dictionary

result2 = dict(a=1,b=2)

# Displaying result

print(result)

print(result2)

Output:

{}

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

Python **filter()** Function

Python filter() function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns true value.

The first argument can be none, if the function is not available and returns only elements that are true.

Example

# Python filter() function example

def filterdata(x):

if x>5:

return x

# Calling function

result = filter(filterdata,(1,2,6))

# Displaying result

print(list(result))

Output:

[6]

Python **help()** Function

Python help() function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

Example

# Calling function

info = help() # No argument

# Displaying result

print(info)

Output:

Welcome to Python 3.8's help utility!

Python **min()** Function

Python min() function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

Example

# Calling function

small = min(2225,325,2025) # returns smallest element

small2 = min(1000.25,2025.35,5625.36,10052.50)

# Displaying result

print(small)

print(small2)

Output:

325

1000.25

Python **set()** Function

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

Python set() Function Example

# Calling function

result = set() # empty set

result2 = set('12')

result3 = set('javatpoint')

# Displaying result

print(result)

print(result2)

print(result3)

Output:

set()

{'1', '2'}

{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}

Python **hex()** Function

Python hex() function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

Python hex() Function Example

# Calling function

result = hex(1)

# integer value

result2 = hex(342)

# Displaying result

print(result)

print(result2)

Output:

0x1

0x156

Python **id()** Function

Python id() function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

Python id() Function Example

# Calling function

val = id("Javatpoint") # string object

val2 = id(1200) # integer object

val3 = id([25,336,95,236,92,3225]) # List object

# Displaying result

print(val)

print(val2)

print(val3)

Output:

139963782059696

139963805666864

139963781994504

Python **setattr()** Function

Python setattr() function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

Example:

class Student:

id = 0

name = ""

def \_\_init\_\_(self, id, name):

self.id = id

self.name = name

student = Student(102,"Sohan")

print(student.id)

print(student.name)

#print(student.email) product error

setattr(student, 'email','sohan@abc.com') # adding new attribute

print(student.email)

Output:

102

Sohan

sohan@abc.com

Python **sorted()** Function

Python sorted() function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

Example

str = "javatpoint" # declaring string

# Calling function

sorted1 = sorted(str) # sorting string

# Displaying result

print(sorted1)

Output:

['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']

Python **next()** Function

Python next() function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

Example

number = iter([256, 32, 82]) # Creating iterator

# Calling function

item = next(number)

# Displaying result

print(item)

# second item

item = next(number)

print(item)

# third item

item = next(number)

print(item)

Output:

256

32

82

Python **input()** Function

Python input() function is used to get an 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 it. It throws an error EOFError if EOF is read.

Example

# Calling function

val = input("Enter a value: ")

# Displaying result

print("You entered:",val)

Output:

Enter a value: 45

You entered: 45

Python **int()** Function

Python int() function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

Example:

# Calling function

val = int(10) # integer value

val2 = int(10.52) # float value

val3 = int('10') # string value

# Displaying result

print("integer values :",val, val2, val3)

Output:

integer values : 10 10 10

Python **isinstance()** Function

Python isinstance() function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The isinstance() function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

Python isinstance() function Example

class Student:

id = 101

name = "John"

def \_\_init\_\_(self, id, name):

self.id=id

self.name=name

student = Student(1010,"John")

lst = [12,34,5,6,767]

# Calling function

print(isinstance(student, Student)) # isinstance of Student class

print(isinstance(lst, Student))

Output:

True

False

Python **oct()** Function

Python oct() function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error TypeError, if argument type is other than an integer.

Example:

# Calling function

val = oct(10)

# Displaying result

print("Octal value of 10:",val)

Output:

Octal value of 10: 0o12

Python **ord()** Function

The python ord() function returns an integer representing Unicode code point for the given Unicode character.

Example:

# Code point of an integer

print(ord('8'))

# Code point of an alphabet

print(ord('R'))

# Code point of a character

print(ord('&'))

Output:

56

82

38

Python **pow()** Function

The python pow() function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

Example:

# positive x, positive y (x\*\*y)

print(pow(4, 2))

# negative x, positive y

print(pow(-4, 2))

# positive x, negative y (x\*\*-y)

print(pow(4, -2))

# negative x, negative y

print(pow(-4, -2))

Output:

16

16

0.0625

0.0625

**Python print() Function**

The python print() function prints the given object to the screen or other standard output devices.

Example:

print("Python is programming language.")

x = 7

# Two objects passed

print("x =", x)

y = x

# Three objects passed

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

Output:

Python is programming language.

x = 7

x = 7 = y

Python **range()** Function

The python range() function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

Example:

# empty range

print(list(range(0)))

# using the range(stop)

print(list(range(4)))

# using the range(start, stop)

print(list(range(1,7 )))

Output:

[]

[0, 1, 2, 3]

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

**Python reversed() Function**

The python reversed() function returns the reversed iterator of the given sequence.

Python reversed() function Example

# for string

String = 'Java'

print(list(reversed(String)))

# for tuple

Tuple = ('J', 'a', 'v', 'a')

print(list(reversed(Tuple)))

# for range

Range = range(8, 12)

print(list(reversed(Range)))

# for list

List = [1, 2, 7, 5]

print(list(reversed(List)))

Output:

['a', 'v', 'a', 'J']

['a', 'v', 'a', 'J']

[11, 10, 9, 8]

[5, 7, 2, 1]

**Python round() Function**

The python round() function rounds off the digits of a number and returns the floating point number.

Python round() Function Example

# for integers

print(round(10))

# for floating point

print(round(10.8))

# even choice

print(round(6.6))

Output:

10

11

7

**Python issubclass() Function**

The python issubclass() function returns true if object argument(first argument) is a subclass of second class(second argument).

Python issubclass() Function Example

class Rectangle:

def \_\_init\_\_(rectangleType):

print('Rectangle is a ', rectangleType)

class Square(Rectangle):

def \_\_init\_\_(self):

Rectangle.\_\_init\_\_('square')

print(issubclass(Square, Rectangle))

print(issubclass(Square, list))

print(issubclass(Square, (list, Rectangle)))

print(issubclass(Rectangle, (list, Rectangle)))

Output:

True

False

True

True

**Python str function:**

The python str() converts a specified value into a string.

str(4)

Output:

'4'

**Python tuple() Function**

The python tuple() function is used to create a tuple object.

Python tuple() Function Example

t1 = tuple()

print('t1=', t1)

# creating a tuple from a list

t2 = tuple([1, 6, 9])

print('t2=', t2)

# creating a tuple from a string

t1 = tuple('Java')

print('t1=',t1)

# creating a tuple from a dictionary

t1 = tuple({4: 'four', 5: 'five'})

print('t1=',t1)

Output:

t1= ()

t2= (1, 6, 9)

t1= ('J', 'a', 'v', 'a')

t1= (4, 5)

**Python type()**

The python type() returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

Python type() Function Example

List = [4, 5]

print(type(List))

Dict = {4: 'four', 5: 'five'}

print(type(Dict))

class Python:

a = 0

InstanceOfPython = Python()

print(type(InstanceOfPython))

Output:

<class 'list'>

<class 'dict'>

<class '\_\_main\_\_.Python'>

**Python Lambda Functions**

Python Lambda function is known as the anonymous function that is defined without a name. Python allows us to not declare the function in the standard manner, i.e., by using the def keyword. Rather, the anonymous functions are declared by using the lambda keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The syntax to define an anonymous function is given below.

Syntax: lambda arguments: expression

It can accept any number of arguments and has only one expression. It is useful when the function objects are required.

Example 1

# a is an argument and a+10 is an expression which got evaluated and returned.

x = lambda a:a+10

# Here we are printing the function object

print(x)

print("sum = ",x(20))

Output:

<function <lambda> at 0x0000019E285D16A8>

sum = 30

In the above example, we have defined the lambda a: a+10 anonymous function where a is an argument and a+10 is an expression. The given expression gets evaluated and returned the result. The above lambda function is same as the normal function.

def x(a):

return a+10

print(sum = x(10))

Example 2

Multiple arguments to Lambda function

# a and b are the arguments and a\*b is the expression which gets evaluated and returned.

x = lambda a,b: a\*b

print("mul = ", x(20,10))

Output:

mul = 200

**Why use lambda function?**

The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function. In Python, the lambda function can be used as an argument to the higher-order functionswhich accepts other functions as arguments.

Example 1:

#the function table(n) prints the table of n

def table(n):

return lambda a:a\*n # a will contain the iteration variable i and a multiple of n is returned at each function call

n = int(input("Enter the number:"))

b = table(n) #the entered number is passed into the function table. b will contain a lambda function which is

#called again and again with the iteration variable i

for i in range(1,11):

print(n,"X",i,"=",b(i)) #the lambda function b is called with the iteration variable i

Output:

Enter the number:10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

**The lambda function is commonly used with Python built-in functions filter() function and map() function.**

**Use lambda function with filter()**

The Python built-in filter() function accepts a function and a list as an argument. It provides an effective way to filter out all elements of the sequence. It returns the new sequence in which the function evaluates to True.

Example:

#program to filter out the tuple which contains odd numbers

lst = (10,22,37,41,100,123,29)

oddlist = tuple(filter(lambda x:(x%3 == 0),lst)) # the tuple contains all the items of the tuple for which the lambda function evaluates to true

print(oddlist)

Output:

(123, )

**Using lambda function with map()**

The map() function in Python accepts a function and a list. It gives a new list which contains all modified items returned by the function for each item.

Example:

#program to filter out the list which contains odd numbers

lst = (10,20,30,40,50,60)

square\_list = list(map(lambda x:x\*\*2,lst)) # the tuple contains all the items of the list for which the lambda function evaluates to true

print(square\_tuple)

Output:

(100, 400, 900, 1600, 2500, 3600)

**Python File Handling**

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the **data needs to be stored permanently into the file**. A file is a named location on disk to store related information. **We can access the stored information (non-volatile) after the program termination.**

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in **two modes as text or binary**. The file may be in the text or binary format, and **each line of a file is ended with the special character.**

Hence, a **file operation can be done in the following order**.

* Open a file
* Read or write - Performing operation
* Close the file
* Opening a file

**open() function:**

Python provides an open() function that **accepts two arguments, file name and access mode** in which the file is accessed. The function **returns a file object which can be used to perform various operations** like reading, writing, etc.

Syntax:

file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the **access mode to open a file.**

|  |  |  |
| --- | --- | --- |
| Sr. No. | Access mode | Description |
| 1 | r | It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| 2 | rb | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file. |
| 3 | r+ | It opens the file to read and write both. The file pointer exists at the beginning of the file. |
| 4 | rb+ | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. |
| 5 | w | It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. |
| 6 | wb | It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file. |
| 7 | w+ | It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file. |
| 8 | wb+ | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. |
| 9 | a | It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name. |
| 10 | ab | It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name. |
| 11 | a+ | It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name. |
| 12 | ab+ | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. |

Example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example:

fileptr = open("file.txt","r")

if fileptr:

print("file is opened successfully")

Output:

<class '\_io.TextIOWrapper'>

file is opened successfully

In the above code, we have passed filename as a first argument and opened file in read mode as we mentioned r as the second argument. The fileptr holds the file object and if the file is opened successfully, it will execute the print statement.

**The close() method**

Once all the operations are done on the file, we must close it through our Python script using the close() method. Any **unwritten information gets destroyed once the close() method is called on a file object**.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

Syntax:

fileobject.close()

Example:

fileptr = open("file.txt","r")

if fileptr:

print("file is opened successfully")

fileptr.close()

**After closing the file, we cannot perform any operation in the file**. The file needs to be properly closed. **If any exception occurs while performing some operations in the file then the program terminates without closing the file.**

**We should use the following method to overcome such type of problem.**

try:

fileptr = open("file.txt")

# perform file operations

finally:

fileptr.close()

**The with statement:**

The with statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

**The syntax to open a file using with the statement is given below.**

with open(<file name>, <access mode>) as <file-pointer>:

#statement suite

**The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.**

It is always suggestible to use the with statement in the case of files because, **if the break, return, or exception occurs in the nested block of code then it automatically closes the file**, we don't need to write the close() function. It doesn't let the file to corrupt.

Example:

with open("file.txt",'r') as f:

content = f.read();

print(content)

**Writing the file (write() method):**

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Example:

# open the file.txt in append mode. Create a new file if no such file exists.

fileptr = open("file2.txt", "w")

# appending the content to the file

fileptr.write('''''Python is the modern day language. It makes things so simple.

It is the fastest-growing programing language''')

# closing the opened the file

fileptr.close()

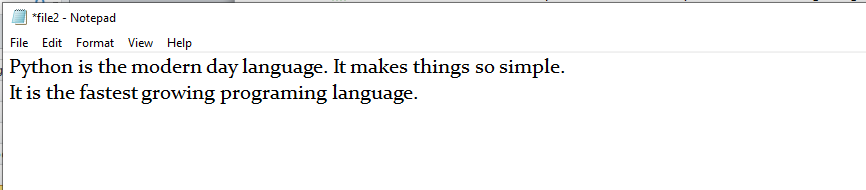
Output:

File2.txt

Python is the modern-day language. It makes things so simple.

It is the fastest growing programming language.

Snapshot of the file2.txt



We have opened the file in w mode. The file1.txt file doesn't exist, it created a new file and we have written the content in the file using the write() function.

Example 2

#open the file.txt in write mode.

fileptr = open("file2.txt","a")

#overwriting the content of the file

fileptr.write(" Python has an easy syntax and user-friendly interaction.")

#closing the opened file

fileptr.close()

Output:

Python is the modern day language. It makes things so simple.

It is the fastest growing programing language Python has an easy syntax and user-friendly interaction.

**Reading the file (read() method):**

To read a file using the Python script, the Python provides the read() method. The read() method reads a string from the file. It can read the data in the text as well as a binary format.

Syntax:

fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Example

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r")

#stores all the data of the file into the variable content

content = fileptr.read(10)

# prints the type of the data stored in the file

print(type(content))

#prints the content of the file

print(content)

#closes the opened file

fileptr.close()

Output:

<class 'str'>

Python is

In the above code, we have read the content of file2.txt by using the read() function. We have passed count value as ten which means it will read the first ten characters from the file.

**If we use the following line, then it will print all content of the file.**

content = fileptr.read()

print(content)

Output:

Python is the modern-day language. It makes things so simple.

It is the fastest-growing programing language Python has easy an syntax and user-friendly interaction.

**Read file through for loop:**

We can read the file using for loop. Consider the following example.

#open the file.txt in read mode. causes an error if no such file exists.

fileptr = open("file2.txt","r");

#running a for loop

for i in fileptr:

print(i) # **i contains each line of the file**

Output:

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

**Read Lines of the file**

Python facilitates to read the file line by line by using a function readline() method. **The readline() method reads the lines of the file from the beginning**, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function readline() that reads the first line of our file "file2.txt" containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content

content = fileptr.readline()

content1 = fileptr.readline()

#prints the content of the file

print(content)

print(content1)

#closes the opened file

fileptr.close()

Output:

Python is the modern day language.

It makes things so simple.

We called the readline() function two times that's why it read two lines from the file.

**readlines() method:**

Python provides also the readlines() method which is used for the reading lines. It returns the list of the lines till the end of file(EOF) is reached.

Example 2: Reading Lines Using readlines() function

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content

content = fileptr.readlines()

#prints the content of the file

print(content)

#closes the opened file

fileptr.close()

Output:

['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']

**Creating a new file:**

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Example 1

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","x")

print(fileptr)

if fileptr:

print("File created successfully")

Output:

<\_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>

File created successfully

**File Pointer positions:**

Python provides the **tell() method which is used to print the byte number at which the file pointer currently exists**.

# open the file file2.txt in read mode

fileptr = open("file2.txt","r")

#initially the filepointer is at 0

print("The filepointer is at byte :",fileptr.tell())

#reading the content of the file

content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())

Output:

The filepointer is at byte : 0

After reading, the filepointer is at: 117

**Modifying file pointer position (seek() method):**

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally. **The seek() method accepts two parameters**:

**offset:** It refers to the new position of the file pointer within the file.

**from:** It indicates the reference position from where the bytes are to be moved. If it is set to **0, the beginning of the file is used as the reference position**. If it is set to **1, the current position of the file pointer is used as the reference position**. If it is set to **2, the end of the file pointer is used as the reference position**.

Syntax:

<file-ptr>.seek(offset[, from)

Example:

# open the file file2.txt in read mode

fileptr = open("file2.txt","r")

#initially the filepointer is at 0

print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.

fileptr.seek(10);

#tell() returns the location of the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())

Output:

The filepointer is at byte : 0

After reading, the filepointer is at: 10

**Python OS module:**

The Python os module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc.

**Renaming the file:**

To rename the specified file to a new name, os module provides rename() method.

Syntax:

rename(current-name, new-name)

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

Example 1:

import os

#rename file2.txt to file3.txt

os.rename("C:\\ Python\\python.txt","C:\\ Python\\test.txt")

Output:

**Removing the file**

The os module provides the **remove() method** which is used to remove the specified file.

remove(file-name)

Example:

import os;

#deleting the file named file3.txt

os.remove("file3.txt")

**Creating the new directory**

The mkdir() method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax:

mkdir(directory name)

Example 1

import os

#creating a new directory with the name new

os.mkdir("new")

**The getcwd() method:**

This method returns the current working directory.

Syntax

os.getcwd()

Example

import os

os.getcwd()

Output:

'C:\\Users\\DEVANSH SHARMA'

**Changing the current working directory:**

**The chdir() method** is used to change the current working directory to a specified directory.

Syntax

chdir("new-directory")

Example

import os

# Changing current directory with the new directiory

os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")

#It will display the current working directory

os.getcwd()

Output:

'C:\\Users\\DEVANSH SHARMA\\Documents'

**Deleting directory:**

The **rmdir() method** is used to delete the specified directory.

Syntax

os.rmdir(directory name)

Example 1

import os

#removing the new directory

os.rmdir("directory\_name")

It will remove the specified directory.

**Writing Python output to the files:**

In Python, there are the requirements to write the output of a Python script to a file.

The **check\_call() method of module subprocess is used to execute a Python script and write the output of that script to a file.**

The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file output.txt.

Example

**file.py**

temperatures=[10,-20,-289,100]

def c\_to\_f(c):

if c< -273.15:

return "That temperature doesn't make sense!"

else:

f=c\*9/5+32

return f

for t in temperatures:

print(c\_to\_f(t))

**file1.py**

import subprocess

f=open("output.txt", "wb")

subprocess.check\_call(["python", "file.py"], stdout=f)

**The file related methods:**

The file object provides the following methods to manipulate the files on various operating systems.

|  |  |  |
| --- | --- | --- |
| SN | Method | Description |
| 1 | file.close() | It closes the opened file. The file once closed, it can't be read or write anymore. |
| 2 | File.fush() | It flushes the internal buffer. |
| 3 | File.fileno() | It returns the file descriptor used by the underlying implementation to request I/O from the OS. |
| 4 | File.isatty() | It returns true if the file is connected to a TTY device, otherwise returns false. |
| 5 | File.next() | It returns the next line from the file. |
| 6 | File.read([size]) | It reads the file for the specified size. |
| 7 | File.readline([size]) | It reads one line from the file and places the file pointer to the beginning of the new line. |
| 8 | File.readlines([sizehint]) | It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function. |
| 9 | File.seek(offset[,from) | It modifies the position of the file pointer to a specified offset with the specified reference. |
| 10 | File.tell() | It returns the current position of the file pointer within the file. |
| 11 | File.truncate([size]) | It truncates the file to the optional specified size. |
| 12 | File.write(str) | It writes the specified string to a file |
| 13 | File.writelines(seq) | It writes a sequence of the strings to a file. |

**Python Modules**

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Example

In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as file.py.

#displayMsg prints a message to the name being passed.

def displayMsg(name)

print("Hi "+name);

Here, we need to include this module into our main module to call the method displayMsg() defined in the module named file.

**Loading the module in our python code**

We need to load the module in our python code to use its functionality. **Python provides two types of statements as defined below.**

* The import statement
* The from-import statement

**The import statement**

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

**We can import multiple modules with a single import statement**, **but a module is loaded once regardless of the number of times, it has been imported into our file**.

Syntax:

import module1,module2,........ module n

import module

Hence, if we need to call the function displayMsg() defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

Example:

import file;

name = input("Enter the name?")

file.displayMsg(name)

Output:

Enter the name?John

Hi John

**The from-import statement:**

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement.

Syntax:

from < module-name> import <name 1>, <name 2>..,<name n>

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

Example:

**calculation.py:**

#place the code in the calculation.py

def summation(a,b):

return a+b

def multiplication(a,b):

return a\*b;

def divide(a,b):

return a/b;

**Main.py:**

from calculation import summation

#it will import only the summation() from calculation.py

a = int(input("Enter the first number"))

b = int(input("Enter the second number"))

print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing summation()

Output:

Enter the first number10

Enter the second number20

Sum = 30

**The from...import statement is always better to use if we know the attributes to be imported from the module in advance.** It doesn't let our code to be heavier. **We can also import all the attributes from a module by using \*.**

Syntax:

from <module> import \*

**Renaming a module:**

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

Syntax:

import <module-name> as <specific-name>

Example

#the module calculation of previous example is imported in this example as cal.

import calculation as cal;

a = int(input("Enter a?"));

b = int(input("Enter b?"));

print("Sum = ",cal.summation(a,b))

Output:

Enter a?10

Enter b?20

Sum = 30

**Using dir() function**

The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Example

import json

List = dir(json)

print(List)

Output:

['JSONDecoder', 'JSONEncoder', '\_\_all\_\_', '\_\_author\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_',

'\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_path\_\_', '\_\_spec\_\_', '\_\_version\_\_',

'\_default\_decoder', '\_default\_encoder', 'decoder', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']

**The reload() function**

As we have already stated that, a module is loaded once regardless of the number of times it is imported into the python source file. However, if you want to reload the already imported module to re-execute the top-level code, python provides us the reload() function. The syntax to use the reload() function is given below.

Syntax:

reload(<module-name>)

for example, to reload the module calculation defined in the previous example, we must use the following line of code.

reload(calculation)

**Scope of variables**

In Python, variables are associated with two types of scopes. All the variables defined in a module contain the global scope unless or until it is defined within a function.

All the variables defined inside a function contain a local scope that is limited to this function itself. We can not access a local variable globally.

**If two variables are defined with the same name with the two different scopes, i.e., local and global, then the priority will always be given to the local variable.**

Example

name = "john"

def print\_name(name):

print("Hi",name) #prints the name that is local to this function only.

name = input("Enter the name?")

print\_name(name)

Output:

Hi David

**Python packages**

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path /home.

2. Create a python source file with name ITEmployees.py on the path /home/Employees.

ITEmployees.py

def getITNames():

List = ["John", "David", "Nick", "Martin"]

return List;

3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is \_\_init\_\_.py which contains the import statements of the modules defined in this directory.

\_\_init\_\_.py

from ITEmployees import getITNames

from BPOEmployees import getBPONames

5. Now, the directory Employees has become the package containing two python modules. **Here we must notice that we must have to create \_\_init\_\_.py inside a directory to convert this directory to a package.**

6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

Test.py

import Employees

print(Employees.getNames())

Output:

['John', 'David', 'Nick', 'Martin']

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.

Python packages

**Python Exception**

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

**Python has many built-in exceptions that enable our program to run without interruption and give the output.** These exceptions are given below:

**Common Exceptions**

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

* ZeroDivisionError: Occurs when a number is divided by zero.
* NameError: It occurs when a name is not found. It may be local or global.
* IndentationError: If incorrect indentation is given.
* IOError: It occurs when Input Output operation fails.
* EOFError: It occurs when the end of the file is reached, and yet operations are being performed.

**The problem without handling exceptions**

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

Suppose we have two variables a and b, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a ZeroDivision exception.

Example

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

print("a/b = %d" %c)

#other code:

print("Hi I am other part of the program")

Output:

Enter a:10

Enter b:0

Traceback (most recent call last):

File "exception-test.py", line 3, in <module>

c = a/b;

ZeroDivisionError: division by zero

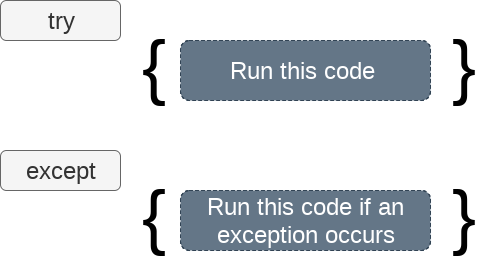
The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions.

We can handle these exceptions in the following way.

**Exception handling in python**

**The try-expect statement**

If the Python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax

try:

#block of code

except Exception1:

#block of code

except Exception2:

#block of code

#other code

Consider the following example.

Example 1

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

except:

print("Can't divide with zero")

Output:

Enter a:10

Enter b:0

Can't divide with zero

**We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.**

Syntax:

try:

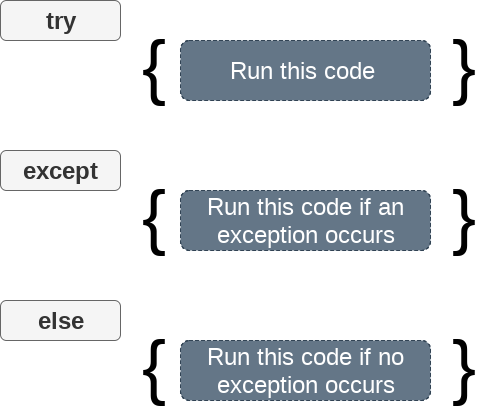
#block of code

except Exception1:

#block of code

else:

#this code executes if no except block is executed



Example 2

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

print("a/b = %d"%c)

# Using Exception with except statement. If we print(Exception) it will return exception class

except Exception:

print("can't divide by zero")

print(Exception)

else:

print("Hi I am else block")

Output:

Enter a:10

Enter b:0

can't divide by zero

<class 'Exception'>

**The except statement with no exception**

Python provides the flexibility not to specify the name of exception with the exception statement.

Example

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b;

print("a/b = %d"%c)

except:

print("can't divide by zero")

else:

print("Hi I am else block")

**The except statement using with exception variable**

We can use the exception variable with the except statement. It is used by using the as keyword. this object will return the cause of the exception.

Example:

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

print("a/b = %d"%c)

# Using exception object with the except statement

except Exception as e:

print("can't divide by zero")

print(e)

else:

print("Hi I am else block")

Output:

Enter a:10

Enter b:0

can't divide by zero

division by zero

**Points to remember**

* Python facilitates us to not specify the exception with the except statement.
* We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
* We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
* The statements that don't throw the exception should be placed inside the else block.

Example

try:

#this will throw an exception if the file doesn't exist.

fileptr = open("file.txt","r")

except IOError:

print("File not found")

else:

print("The file opened successfully")

fileptr.close()

Output:

File not found

**Declaring Multiple Exceptions**

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.

Syntax

try:

#block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)

#block of code

else:

#block of code

Example:

try:

a=10/0;

except(ArithmeticError, IOError):

print("Arithmetic Exception")

else:

print("Successfully Done")

Output:

Arithmetic Exception

**The try...finally block**

Python provides the optional finally statement, which is used with the try statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

Syntax

try:

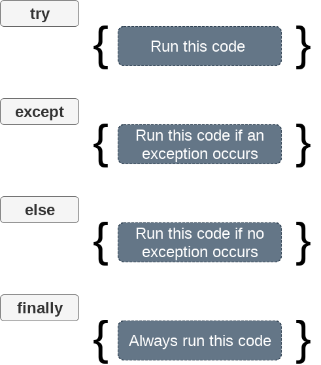
# block of code

# this may throw an exception

finally:

# block of code

# this will always be executed



Example

try:

fileptr = open("file2.txt","r")

try:

fileptr.write("Hi I am good")

finally:

fileptr.close()

print("file closed")

except:

print("Error")

Output:

file closed

Error

**Raising exceptions**

An exception can be raised forcefully by using the raise clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

Syntax

raise Exception\_class,<value>

**Points to remember**

* To raise an exception, the raise statement is used. The exception class name follows it.
* An exception can be provided with a value that can be given in the parenthesis.
* To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.
* We can pass the value to an exception to specify the exception type.

Example

try:

age = int(input("Enter the age:"))

if(age<18):

raise ValueError

else:

print("the age is valid")

except ValueError:

print("The age is not valid")

Output:

Enter the age:17

The age is not valid

Example 2 Raise the exception with message

try:

num = int(input("Enter a positive integer: "))

if(num <= 0):

# we can pass the message in the raise statement

raise ValueError("That is a negative number!")

except ValueError as e:

print(e)

Output:

Enter a positive integer: -5

That is a negative number!

Example 3

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

if b is 0:

raise ArithmeticError

else:

print("a/b = ",a/b)

except ArithmeticError:

print("The value of b can't be 0")

Output:

Enter a:10

Enter b:0

The value of b can't be 0

**Custom Exception**

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

Example

class ErrorInCode(Exception):

def \_\_init\_\_(self, data):

self.data = data

def \_\_str\_\_(self):

return repr(self.data)

try:

raise ErrorInCode(2000)

except ErrorInCode as ae:

print("Received error:", ae.data)

Output:

Received error: 2000