# Introduction of Programming

## Basic Commands

1. Create New Folder
   mkdir New_Videos_2023
2. Remove Folder
   rm -rf New_Videos_2023
3. Create New File
   touch New_Videos_2023.py
4. Remove File
   rm New_Videos_2023.py
5. Go To Directory
   cd f://
6. Go Back To Folder
   cd..
7. To See Working Directory
   pwd
8. To See List
   ls
9. To Code A File
   code file_name.extension
10. To Open Terminal
    Ctrl+~
11. To Run Code
    Python file_name.extension
12. To Clear Screen
    clear
13. To Rename A File
    mv old_file.extension new_file.extension
14. To Move A File
    mv old_file.extension bb
15. To Copy A File
    cp old_file.extension path_tocopy

## Concept of Software (Programming)

Software (programming) rules the hardware (the physical machine). A program is just a sequence of instructions telling a computer what to do. The process of creating software is called programming.

## How to write a program?

By designing notations for expressing computations in an exact and unambiguous way (word that have only one meaning). These special notations are called programming languages. Every structure in a programming language has a precise form (its syntax) and a precise meaning(its semantics).

# Types of programming

1. Low level of programming language
2. High level programming language

## Low level of programming language

1. Machine Dependent
2. Hardware Interactive
3. Difficult to write program
4. Difficult to debug program
5. Fast in execution
6. Example: machine language and Assembly language

## High level programming language

1. Machine Independent
2. No Hardware Interactive
3. Easy to write program
4. Easy to debug program
5. Slow in execution
6. Example: C, C++, JAVA,Python

# Translator

That’s a lot easier for us to understand and write programs in High Level Language (HLL), but we need some way to translate the high-level language into the machine language that the computer can execute. There are two types of translators: compiler and interpreter.

## Compiler

A compiler is a computer program that takes a program written in a high-level language and translates it into an equivalent program in the machine Understandable format that the computer can directly execute. Need for an executor or system to run the machine code.

### Interpreter

An interpreter is a program that simulates ( نقل کرتا ہے ) a computer that understands a high-level language. Rather than translating the source program into a machine language equivalent, the interpreter analyses and executes the source code instruction by instruction as necessary. No need for executor or system to run machine code.

# History of Python

Guido van Rossum, Python's creator, started developing Python back in 1990. The language was finally released in 1991. Python is named after the British comedy group Monty Python.

| Versions | Release Date |
|---|---|
| 0.9.0 | February 1991 |
| 1.0 | January 1994 |
| 2.0 | October 2000 |
| 3.0 | December 2008 |

Latest version of python is 3.12.1.

Python Software Foundation (PSF) used to support two major versions, Python 2.x & Python 3.x. PSF supported Python 2 because a large body of existing code could not be forward ported to Python 3. So, they supported Python 2 until January 2020, but now they have stopped supporting it.

# Who Uses Python Today?

1. Google makes extensive use of Python in its web search systems.

2. The popular YouTube video sharing service is largely written in Python.

3. The Dropbox storage service codes both its server and desktop client software primarily in Python.

4. The Raspberry Pi Single-board computer promotes Python as its educational language.

5. EVE Online, a massively multiplayer online game (MMOG) by CCP Games, uses Python broadly.

6. The widespread BitTorrent peer-to-peer file sharing system began its life as a Python program.

7. Industrial Light 8 Magic, Pixar, and others use Python in the production of animated movies.

8. ESRI uses Python as an end-user customization tool for its popular GIS mapping products.

# What are Python's Technical Strenth?

1. it's Relatively Easy to Use

2. it's Free

3. It's Portable

4. lt's General Purpose Language

5. lt's Powerful

6. Dynamic typing

7. Automatic memory management

8. Programming-in-the-large support

9. Built-in object types

10. Built-in operation

11. Library utilities & Third-party utilities

12. lt's Mixable

13. It's Object-Oriented and Functional

# Popular Python Implementations

1. CPython

2. Jython

3. IronPython
4. PyPy

Base: Programming Language and Running Environment (Virtual Machine).

## Popular Python Implementations

| | | | |
|---|---|---|---|
| Cpython | Jython | IronPython | PyPy |
| 1994 | 2001 | 2006 | 2007 |
| C Language | Java | C# | Rpython |
| Cpython VM | JVM | NET or CLR | JIT |

# Is Python Already Present?

Before you do anything else, check whether you already have recent Python on your machine. If you are working on Linux, Mac OS X, or some Unix systems, Python is probably already installed on your computer.

1. Check using CMD (python –V)
2. Check Directories in C Drive
3. Check Install Program List
4. Check Through Start Menu

# Where to Get Python

If there is no Python on your machine, you will need to install one yourself. The good news is that Python is an open source system that is freely available on the Web and very easy to install on most platforms. Ater download we get Python Interpreter or Python Virtual Machine (PVM) and Tools(IDLE + Shell) etc.

http://www.python.org

# Configuring Python

After you've installed Python, you may want to configure some system settings that impact the way Python runs your code. (If you are just getting started with the language, you can probably skip this section completely: there is usually no need to specify any system settings for basic programs.)

# After Installation

Check Python installation is working fine or not, we check version of python

Python –V

# Python Interpreter / Python Virtual Machine (PVM)

Python Interpreter is a program that simulates ( نقل کرتا ہے ) a computer that understands a high-level language. Rather than translating the source program into a machine language equivalent, the interpreter analyses and executes the source code instruction by instruction as necessary.

# Program Execution

**The Programmer's View :**

A Python program is just a text file containing Python statements. Python program files are given names that end in .py . You must tell Python to execute the file—which simply means to run all the statements in the file from top to bottom, one after another. Python first compiles your source code (the statements in your file) into a format known as **byte code**.

**The PVM's View :**


Compilation is simply a translation step, and byte code is a lower-level, platform-independent representation of your source code. Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This bytecode translation is performed to speed execution—byte code can be run much more quickly than the original source code statements in your text file.

Python bytecode is not binary machine code (e.g., instructions for an Intel or ARM chip). Byte code is a Python-specific representation.Python will store the byte code of your programs in files that end with a .pyc extension (".pyc- means compiled ".py" source.

In 3.2 and later Python instead saves its .pyc bytecode files in a subdirectory named _pycache_ located in the directory where your source files reside, and in files whose names identify the Python version that created them (e.g., script.cpython-33.pyc).

Python simply creates and uses the byte .file in memory and discards it on exit. To speed startups, though, it will try to save byte code in a file in order to skip the compile step next time around.The next time you run your program.

Python will load the .pyc files and skip the compilation step, as long as you '.haven't changed your source code since the byte code was last saved, and aren't running with a different Python than the one that created the byte code.Once your program has been compiled to bytecode (or the byte code has been loaded from existing .pyc files),it is shipped off for execution to something generally known as the Python Virtual Machine.

Byte code compilation is automatic, and the PVM is just part of the Python system that you have installed on your machine. Again, programmers simply code and run files of statements, and Python handles the logistics of running them.

In Python is runtime-there is no initial compile-time phase at all, and everything happens as the program is running.

# Token/lexical token

Set of characters is called a token. Tokens are unbreakable. There are five types of tokens:

1. Keywords
2. Identifiers
3. Punctuators/ Punctuation characters
4. White Spaces/ indentation
5. Literal/values/data/constant
    a. Number
        i. Integer
            1. Decimal
            2. Octol
            3. Hexadecimal
            4. Binary number
        ii. Float Number/ floating point Number
        iii. Complex Number
    b. Boolean

c. None

d. String

## Keywords

Keywords are reserved words and predefined in python. They have special meaning in python.

| | | | | | | |
|---|---|---|---|---|---|---|
| If | elif | Else | for | while | break | continue |
| Pass | async | Await | yield | and | or | not |
| True | False | Try | except | finally | class | def |
| Return | None | From | import | in | global | nonlocal |
| is | as | With | assert | del | lambda | raise |

## Identifiers

An identifier is a name used to identify a class, Method, variable or any other user-defined types. Some rules followed for declaring identifiers:

A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.

It must not contain any embedded space or symbol such as? - + ! @ # % ^ & * ( ) [ ] { } . : “‘/ and \. However, an underscore (_) can be used.

## Punctuators

Punctuators are predefined symbols in python that have special meaning in python. For example +, -, *, /, <, >,(), {}, [] etc.

## White Spaces/indentation

Indentation refers to the spaces at the beginning of a code line. Python uses indentation to indicate a block of code.

## Literal/values/data/constant

Literals are data or values on which our programs works.Literls are constant.

## Number

Numeric literals can include single underscore (_) characters between digits or after any base specifier. underscore(_) use in place of comma(,).This is used for programmers not for user . As this implies, not only decimal numeric constants can benefit from this new notational freedom:

>>> 100_000.000_0001, 0x_FF_FF, 007_777, 0b_1010_1010 (100000.0000001, 65535, 4095, 170)

Python provides a special function called type() that tells us the data type (or "class") of any value.

### Integer

Integers may be coded in four forms: decimal, hexadecimal, octal and binary.

#### decimal (base 10)

decimals digits starts from 0 to 9.

#### hexadecimal (base 16)

Hexadecimals start with a leading Ox or OX, followed by a string of hexadecimal digits (0-9 and A-F). Hex digits may be coded in lower- or uppercase.

#### octal (base 8)

Octal literals start with a leading Oo or 00 (zero and lower- or uppercase letter o), followed by a string of digits (0-7).

#### binary (base 2)

Binary literals begin with a leading Ob or OB, followed by binary digits (0-1).

**Note:**

The built-in calls hex(1), oct(l), and bin(1) convert an integer to its representation string in these three bases, and int(str, base) converts a runtime string to an integer per a given base.

#### Integer Methods

Integer methods are given below.

##### bin() Method

Return the binary representation of an integer.

| | |
|---|---|
| Parameters | (number: Union[int, _SupportsIndex], /) |
| Return | string |

#### hex() Method

Return the hexadecimal representation of an integer.

| | |
|---|---|
| Parameters | (number: Union[int, _SupportsIndex], /) |
| Return | string |

#### oct() Method

Return the octal representation of an integer.

| | |
|---|---|
| Parameters | (number: Union[int, _SupportsIndex], /) |
| Return | string |

#### Floating

Floats are numbers with a decimal point, like 2.376, -99.1, and 1.0. We know $2*10^{10}$=20000000000. In python we can write this in scientific form 2e10.

#### Imaginary/Complex Number

Python complex literals are written as real part+imaginary part, where the imaginary part is terminated with a j or J. The real part is technically optional, so the imaginary part may appear on its own. Complex numbers may also be created with the complex(real, imag) built-in call.

## Boolean

The Python Boolean type is one of Python's built-in data types. It's used to represent the truth value of an expression. For example, the expression 1 <= 2 is True , while the expression 0 == 1 is False . Understanding how Python Boolean values behave is important to programming well in Python.

## None

NoneType in Python is a data type that simply shows that an object has no value/has a value of None . You can assign the value of None to a variable but there are also methods that return None .

## String

Collection of characters in single quotes(‘ ’) or double quotes(‘ ’) or triple quotes(‘ ’) is called string literals. A  string is a collection of characters, which is ordered , indexed, immutable, iterable and allows duplicate values. Nested Structure is not allowed.

1. Single quotes: 'spa"m'
2. Double quotes: "spa'm"
3. Triple quotes: "... spam..."

Strings in Python are immutable To have a string literal span multiple physical lines, you can use a as the last character of a line to indicate that the next line is a continuation: 'A not very long string \ that spans two lines'.

**Escape sequences (This is used for non-print able characters)**

Backslashes are used to introduce special character in coding known as **escape sequences**. Escape sequences let us embed characters in strings that cannot easily be typed on a keyboard.

The character \, and one or more characters following it in the string literal, are replaced with a single character in the resulting string. For example, here is a five-character string that embeds a newline and a tab: >>> s'a\nb\tc' **len function**--it returns the actual number of characters in a string. A few escape sequences only work as advertised if you run your program directly from the operating system and not through IDLE. The escape sequence la is a good example

\\ Backslash (stores one \)

\' Single quote (stores ')

\" Double quote (stores ")

\a bell(this sequence is not work on IDLE)

\b Backspace

\f Formfeed (this sequence is not work on IDLE). .Use when use Printer

\n Newline (linefeed)

\r Carriage return

\t Horizontal tab

\v Vertical tab(this sequence is not work on IDLE).Use when use Printer

**ord() for number and chr() for character**

\xhh Character with hex value hh (exactly 2 digits) e.i for tab ( x09)

\ooo Character with octal value ooo (up to 3 digits) e.i for tab ( 011 )

\0 Null: binary 0 character (doesn't end string)

\N{id} Unicode database ID print("Name \t \N{INDIAN RUPEE SIGN}")

\uhhhh Unicode character with 16-bit hex value

\Uhhhhhhhh Unicode character with 32-bit hex valuea

\other Not an escape (keeps both \ and other)

In Python, a zero (null) character like this does not terminate a string the way a "null byte" typically does in C. Instead, Python keeps both the string's length and text in memory. In fact, no character terminates a string in Python.Python does not support a single character constant.

# Reference/Variable

A Python program accesses data values through **references**.A reference is a **"name"** that refers to a value (object).You can name a variable with any identifier except the Python's keywords.In Python, a variable or other reference has no **intrinsic type**.The object to which a reference is bound at a given time always has a type, but a given reference may be bound to objects of various types in the course of the program's execution.Using the id() function, you can verify that two variables indeed point to the same object.

## Name of Variable

A reference is a **"name"** that refers to a value (object). You can name a variable with any **identifier** except the Python's keywords.

**Camel Case:**

Example: numberOfCollegeGraduates

**Pascal Case:** Identical to Camel Case, except the first word is also capitalised. Example: NumberOfCollegeGraduates

**Snake Case:** Words are separated by underscores. Example: number_of_college_graduates

## Binding & Unbinding of Reference

In Python, there are no **"declarations."**The existence of a reference begins with a statement that **binds** the reference (in other words, sets a name to hold a reference to some

object).You can also **unbind** a reference, resetting the name so it no longer holds a reference.Assignment statements are the most common way to bind reference.The **del statement** unbinds references.

## Variables goes into GC

Binding a reference that was already bound is also known as **rebinding** it.Binding or **unbinding** a reference has no effect on the object to which the reference was bound, except that an object goes away when nothing refers to it.The cleanup of objects with no references is known as garbage collection.

## Assignment Statements

Assignment statements can be **plain** or **augmented**.**Plain assignment** to a variable (e.g., name=value) is how you create a new variable or rebind an existing variable to a new value. **Augmented assignment** (e.g.,name=name+value) creates new references.Augmented assignment can rebind a variable.

## Plain Assignment

A plain assignment statement in the simplest form has the syntax:

**Variable = value | expression**

A plain assignment can use multiple targets and equals signs (=). For example:

**a=b=c=0**

The target in a plain assignment can list two or more references separated by commas, optionally enclosed in parentheses or brackets. For example:

**a, b, c = 3 #error**

**a, b, c=5,6,7 #ok**

**a, b, *c=5,6,7,8 #ok**

This kind of assignment is known as an **unpacking assignment**.

## Augmented assignment

An augmented assignment (sometimes also known as an in-place assignment) differs from a plain assignment in that, instead of an equals sign (=) between the target and the expression, it uses an augmented operator, which is a binary operator followed by =.The augmented operators are +=, -=, *=, |=, //=, %=, **=, |=, >>=, <<=, &=, ^=, and @=. An augmented assignment can have only one target on the LHS; augmented assignment doesn't support multiple targets.

# Data Type

The operation of a Python program hinges on the data it handles.Data values in Python are known as objects; each object, AKA value, has a type. An **object's type** determines which type **operations the object supports** (in other words, which operations you can perform on the value).

## type() and isinstance()

The built-in type(obj) accepts any object as its argument and returns the type object that is the type of obj.The built-in function isinstance(obj, type) returns True when object obj has type type (or any subclass thereof); otherwise, it returns False.

## The Dynamic Typing Interlude

Python is dynamically typed, a model that keeps track of types for you automatically instead of requiring declaration code.In Python, types are determined automatically at runtime, not in response to declarations in your code.

## Variants of Data Types

1. Pre-defined or build-in data type
2. User-defined data type

## Data Type in Python

1. int
2. float
3. complex
4. str
5. bool
6. set
7. frozenset
8. tuple
9. list
10. dictionary

11. file
12. function
13. nonetype
14. bytes, bytearray, memoryview
15. classes

## References

| | |
|---|---|
| x = "Hello World" | str |
| X = 20 | int |
| X = 20.5 | float |
| X = 1j | complex |
| X =["apple", "banana", "cherry"] | list |
| x = ("apple", "banana", "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" | |
| x = bytearray(5) | |
| X = memoryview(bytes(5)) | |

## Type Casting

| | |
|---|---|
| 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) | |
| x = bytearray(5) | |
| x = memoryview(bytes(5)) | |

# Frozen Binaries

With the help of third-party tools your Python programs into true executables, known as **frozen binaries** in the Python world.These programs can be run without requiring a Python installation.Frozen binaries bundle together the **byte code** of your program files, along with the **PVM** (interpreter) and any **Python support files** your program needs, into a single package. The end result can be a single **binary executable program** (e.g., an .exe file on

Windows) that can easily be shipped to customers.Today, a variety of systems are capable of generating frozen binaries, which vary in platforms and features:

1. **py2exe** for **Windows** only, but with broad Windows support;
2. Pylnstaller, which is similar to py2exe but also works on Linux and Mac OS X.
3. **py2app** for creating Mac OS X applications.
4. **freeze**, and **cx_freeze**, which offers both Python 3.X and cross-platform support.

# How to Run Python Program

1. Interactive Mode
    a. Using cmd
    b. Using Python shell
    c. Using IDLE or other GUI ID
2. Script Mode

## Interactive Mode

### using cmd

(a) open cmd in your system

(b) Typing the word "python" or "py" or "py -3.9" at your system shell prompt like this begins an interactive Python session

(c) On Windows, a Ctrl-Z gets you out of this session

### Using python shell

(a) Open python cell using run commands then type py then enter

(b) Search python and click on it

### Using IDLE or other GUI ID

In which we use IDE( integrated development environment).

## Why the Interactive Prompt?

1. Experimenting

2. Testing

## The Interactive Mode/Prompt

1. Type Python commands only.
2. print statements are not required most of the times
3. Don't indent at the interactive prompt (yet).
4. Terminate compound statements
5. The interactive prompt runs one statement or multiple statements
6. Entering multi line statements are allowed
7. Program not stored

## Script/ File mode

### Running Files with Command Lines

1. Open notepad and write python program
2. Open cmd
3. Execute python program by command

**python script1.py**

### Command-Line Usage Variations

a. Save Output in Hard Disk

**$ py -3.12 python_test.py > python_test_output_python_version.txt**
**$ python python_test.py > python_test_output.txt**
**$ py python_test.py > python_test_output.txt**

b. See code on console, not work on bash use cmd

**type python_test.py**

c. Create .pyc file

**python**
**import python_test**

### Using IDLE

Besides command history and syntax colorization) IDLE has additional usability features such as:

1. Auto-indent and unindent for Python code in the editor (Backspace goes back one level)
2. Word auto-completion while typing, invoked by a Tab press

3. Balloon help pop ups for a function call when you type its opening "("

4. Pop-up selection lists of object attributes when you type a "." after an object's name and either pause or press Tab

### Usage Notes: Command Lines and Files

1. Beware of automatic extensions
2. Use file extensions and directory paths at system prompts
3. Use print statements
4. Program saved permanently

# Python Operators

Operators are used to perform operations on variables and values.Types of python Operators are given below:

1. Arithmetic operators
2. Comparison operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Identity operators
7. Membership operators

## Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations. They are may be Uniery, Binary and turnery operators.

| **Operator** | **Name** |
|---|---|
| + | Addition |
| - | Subtraction |

| | |
|---|---|
| * | Multiplication |
| / | Division |
| % | Modulus |
| ** | Exponentiation |
| // | Floor division |

### Working with other data types

| Operator | Operator Name | Number | String | Boolean |
|---|---|---|---|---|
| + | Unary plus | Yes | No | Yes |
| - | Unary minus | Yes | No | Yes |
| + | Addition | Yes | Yes | Yes |
| - | Subtraction | Yes | No | Yes |
| * | Multiplication | Yes | Yes with Number | Yes |
| / | Division | Yes | No | Yes |
| % | Modulus | Yes | No | Yes |
| ** | Exponentiation | Yes | No | Yes |
| // | Floor division | Yes | No | Yes |

## Comparison Operators

Comparison operators are used to compare two values or reference. They are Binary Operators and output True and False,

| Operator | Name |
|---|---|
| == | Equal |
| != | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

### Working with other data types

| Operator | Operator Name | Number | String | Boolean |
|---|---|---|---|---|
| == | Equal | Yes | Yes | Yes |
| != | Not equal | Yes | Yes | Yes |
| > | Greater than | Yes | Yes | Yes |
| < | Less than | Yes | Yes | Yes |
| >= | Greater than or equal to | Yes | Yes | Yes |
| <= | Less than or equal to | Yes | Yes | Yes |

## Logical Operators

Logical operators are used to combine conditional statements. 0, False, “” is false

| Operator | Description |
|---|---|
| And | Returns True if both statements are true |
| Or | Returns True if one of the statements is true |
| Not | Reverse the result, returns False if the result is true |

### Working with other data types

| Operator | Boolean | Number | String |
|---|---|---|---|
| Not | Yes | Yes | Yes |
| And | Yes | Yes | Yes |
| Or | Yes | Yes | Yes |

## Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

| Operator | Name | Description |
|---|---|---|
| & | AND | Sets each bit to 1 if both bits are 1 |

| | | |
|---|---|---|
| \| | OR | Sets each bit to 1 if one of two bits is 1 |
| Λ | XOR | If same give 0 Other wise give 1 |
| ~ | NOT or 1st complement | Inverts all the bits unier operator<br>MSB(Most Significient Bit) LSB (Least Significent Bit)<br>1. 1st complement<br>2. put – if MSB is 1<br>3. Tale 2nd complement and add 1<br>4. convert bits into decimle |
| << | Left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off |
| >> | Right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off<br>MSB rule: if number is positive MSB fill by zero otherwise fill by 1. |

## Working with other data types

| Operator | Integer | Float | String | Boolean |
|---|---|---|---|---|
| & | Yes | No | No | Yes |
| \| | Yes | No | No | Yes |
| ^ | Yes | No | No | Yes |
| ~ | Yes | No | No | Yes |

| | | | | |
|---|---|---|---|---|
| << | Yes | No | No | Yes |
| >> | Yes | No | No | Yes |
| | | | | |

## Assignment Operators

Assignment operators are used to assign reference to value

| **Operator** | **Example** | **Same As** |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x+=3 | x = x +3 |
| _= | x-=3 | x=x-3 |
| *= | X*=3 | x=x*3 |
| /= | x/=3 | x=x/3 |
| %= | X%=3 | x=x%3 |
| //= | x//=3 | x=x//3 |
| **= | X**=3 | x=x**3 |
| &= | x&=3 | x=x&3 |
| \|= | x\|=3 | x=x\|3 |
| ^= | x^=3 | x=x^3 |
| >>= | x>>=3 | x=x>>3 |

| | | |
|---|---|---|
| <<= | x<<=3 | x=x<<3 |

## Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

| Operator | Description | Example |
|---|---|---|
| Is | Returns True if both variables are the same object | x is y |
| is not | Returns True if both variables are not the same object | x is not y |

### Identity Operators

| Operator | Integer | Float | String | Boolean |
|---|---|---|---|---|
| Is | Yes | Yes | Yes | Yes |
| is not | Yes | Yes | Yes | Yes |

## Membership Operators

Membership operators are used to test if a sequence is presented in an object.

| Operator | Description | Example |
|---|---|---|
| In | Returns True if a sequence with the specified value is present in the object | x in y |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |

### Working with other data types

| Operator | Integer | Float | String | Boolean |
|---|---|---|---|---|
| In | Yes | Yes | Yes | Yes |
| in not | Yes | Yes | Yes | Yes |

# print() in python

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 being written to the screen.

**Signature Syntax**

**print(", sep=", end='\n', file=sys.stdout, flush=False)**

**print() return None.**

Calling print()

**print()**

## Parameters in print()

| **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). is False |

## Blank Line

print()

or

**print(", sep=", end='\n', file=sys.stdout, flush=False)**

## Printing String through print()

print("Hello")

or

**print('hello', sep=", end='\n', file=sys.stdout, flush=False)**

print('Hello')

print("Hello"Hi')

print('Hello'+'Hi')

print('Hello','HI')

## Separator in print()

>>> print('hello', 'world', sep=None)

**hello world**

>>> print('hello', 'world', sep=' ')

**hello world**

>>> print('hello', 'world')

**hello world**

>>> print('hello', 'world', sep='\n')

**hello World**

>>> print('home', 'user', 'documents', sep='/')

**home/user/documents**

# Separator in print()

>>> print(*['jdoe is', 42, 'years old'])

jdoe is 42 years old

>>> print(1, 'Python Tricks', 'Dan Bader', sep=',')

1,Python Tricks, Dan Bader

>>> print('node', 'child', 'child', sep=' -> ')

node-> child -> child

# End in print()

print('Printing in a Nutshell', end= '\n*')

print('Calling Print', end='\n * ')

print('Separating Multiple Arguments', end='\n*')

print('Preventing Line Breaks')

# File in print()

print('Hello')

or

print('Hello', sep='', end='\n', file=sys.stdout, flush=False)

# input() in python

This function first takes the input from the user. Accept only String from the user.

**Signature Syntax**

**Input(prompt)**

Prompt: Optional:

A string, representing a default message before the input.

**Calling print()**

Input()

# Precedence and Associativity of Operators

To evaluate these types of expressions there is a rule of precedence in Python. It guides the order in which these operations are carried out.

| **Operator** | **Associativity** |
|---|---|
| ( ) | left-to-right |
| ** | right-to-right |
| * / % | left-to-right |
| + - | left-to-right |
| << >> | left-to-right |
| < <=<br>> >= | left-to-right |
| == != | left-to-right |

| | |
|---|---|
| is, is not<br>in, not in | left-to-right |
| & | left-to-right |
| ^ | left-to-right |
| \| | left-to-right |
| not | right-to-left |
| and | left-to-right |
| or | left-to-right |
| =<br>+= -=<br>*= /=<br>%= &=<br>^= \|=<br><<= >>= | right-to-left |

When two operators have the same precedence, associativity helps to determine the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity.

# Non associative operators

Some operators like assignment operators and comparison operators do not have associativity in Python. There are separate rules for sequences of this kind of operator and cannot be expressed as associativity.

For example, x < y < z neither means (x < y) < z nor x < (y <z). x < y <z is equivalent to x < y and y < z, and is evaluated from left-to-right. 2<2<1

Furthermore, while chaining of assignments like x = y = z = 1 is perfectly valid, x = y = z+= 2 will result in

error.

# Get Multiple inputs From a User in One Line

In Python, It is possible to get multiple values from the user in one line. We can accept two or three values from the user.

Take each input separated by space. Split input string using split() get the value of individual input

# Python Casting: Type Conversion and Type Casting

In Python, we can convert one type of variable to another type. This conversion is called type casting or type conversion.

# types of casting

**Implicit casting:** The Python interpreter automatically performs an implicit Type conversion, which avoids loss of data.

**Explicit casting:** The explicit type conversion is performed by the user using built-in functions.

To perform a type casting, we are going to use the following built-in functions

**int():** convert any type variable to the integer type.

**float():** convert any type variable to the float type.

**complex():** convert any type variable to the complex type.

**bool():** convert any type variable to the bool type.

**str():** convert any type variable to the string type.

# Int type conversion

In int type conversion, we use the int() function to convert variables of other types to int type. Variables can be of any type such as float, string, bool.While performing int type conversion, we need to remember the following points. When converting string type to int type, a string must contain integral value only and should be base-10. We can convert any type to int type, but we cannot perform complex to int type.

# Float type conversion

In float type conversion we use a built-in function float(). This function converts variables of other types to float types. While performing float type conversion, we need to remember some points. We can convert any type to float type, but we cannot cast complex to float type. While converting string type to float type, a string must contain an integer/decimal value of base-10.

# Complex type conversion

In complex type conversion, we use the built-in function complex() to convert values from other types to the complex type. Value can be any type including int, float, bool, str. The complex function has the following two forms for conversion.

**complex(x):** To convert a value x into a complex type. In this form, the real value is x, and the imaginary **value is 0. complex(x, y):** To convert the value x and y into a complex type. In this form, the real value is x, and the imaginary is y.

# bool type conversion

we use the built-in function bool() to convert values of other types to bool types. This function returns two values, either True and False.We can convert any type of values to bool type, and the output for all values will be True, Except 0, which is False. If you convert an empty string to a boolean it will be converted to boolean False. The bool True is 1 and False is 0. Every non-zero value is treated as True.

# String type conversion

In str type conversion, we use the built-in function_str() to convert variables of other types to a string type. This function returns the string type of object (value).

# Formatted Output

In general, you will want to have more formatting control over the output of your program than simply printing a space- separated value.

1. Better representation
2. More options over printing

# Ways to Formatted Output

There are several ways to format output.

1. Formatted String Literals
2. Format()

# Formatted string Literals Like C

much like a printf()-style format as in **C language** Formatting output using String modulo operator %,called a **string modulo or format operator**.

**print(f"Additoin of %d and %d is = %d" % (5,6,5+6))**

# Formatted Conversion specifiers

**%[<flags>][<width>][.<precision>]<type>**

| Component | Meaning |
| --- | --- |
| % | Introduces the conversion specifier |
| <flags> | Indicates one or more flags that exert finer control over formatting |

| | |
|---|---|
| <width> | Specifies the minimum width of the formatted result |
| <precision> | Determines the length and precision of floating p or string output |
| <type> | Indicates the type of conversion to be perform |

# Type specifiers

**%[<flags>][<width>][.<precision>]<type>**

| String Formatting Conversion Characters | |
|---|---|
| Character | Output Format |
| d,I,u | Decimal integer |
| x,X | Hexadecimal integer |
| o | Octal integer |
| f,F | Floating point |
| e,E | Exponential |
| g,G | Floating point or Exponential |
| C | Single character |
| s,r,a | String |

| | |
|---|---|
| % | Single ‘%’ character |

## point values

| Character | Controls |
|---|---|
| # | Display of base or decimal point for integer and floating point values |
| 0 | Padding of values that are shorter than the specified field width |
| - | Justification of values that are shorter than the specified field width |
| + | Display of leading sign for numeric values |

## Examples

print('Hello, my name is %s.' % 'Graham')

print("%d %s cost $%.2f' % (6, 'bananas', 1.74))

## Integer Conversion Types

The d, i, u, x, X, and o conversion types correspond to integer values.

print(f"a=%d"%65)

print(f"a=%o%65)

print(f"a=%X"%65)

d, i, and u are functionally equivalent.

## Floating Point Conversion Types

The f,Fg,G,e and E conversion types correspond to Floating values.

print(f"a=%f"%6.5)

print(f"b=%F"%6.5)

print(f"c=%g"%6.5)

print(f"d=%G"%6.5)

print(f"e=%e"%6.5)

print(f"f=%E"%6.5)

# character conversion types

The c conversion types correspond to character values.

The c conversion type supports conversion to Unicode characters as well

print(f"a=%c"%'A')

print(f"b=%c"%65)

print(f"a=%c"%"\u20B9')

# String Conversion Types

s,r, and a produce string output using the built-in functions str(), repr(), and ascii(), respectively

print(f"a=%s"%'Hello')

print(f"b=%r"%'Hello')

print(f"c=%a"%'Hello')

# Insert % character

print(f"a=%d%%"%90)

# The <width>Specifier

<width> specifies the minimum width of the output field. If the output is shorter than <width>, then by default it is right- justified in a field that is <width> characters wide, and padded with ASCII space characters on the left.

print(f"a=%5d"%65)

print(f"b=%5f"%6.5)

print(f"c=%5c"%'A')

print(f"d=%5s"%'hi')

# The .<precision> Specifier

.<precision> affects the floating point, exponential, and string conversion types. For the f, F, g,G,e, and E types, .<precision> determines the number of digits after the decimal point. String values formatted with the s, r, and a types are truncated to the length specified by .

print(f"a=%.2d"%65)

print(f"b=%.2f"%6.5)

print(f"c=%.2c"%'A')

print(f"d=%.2s"%'hello')

# Conversion Flags (#,0,+,-, ' ')

The # Flag: For the octal and hexadecimal conversion types.

The 0 Flag: causes padding with '0' in numbers

The Flag: When a formatted value is shorter than the specified field width, it is usually right-justified.

The + and ''Flags

I By default, positive numeric values do not have a leading

sign character. The + flag adds a '+' character to the left of numeric output.

# Formatted String Literals Python

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with f or F and writing expressions as {expression}.

print(f"Additoin of {2} and {3} is = {2+3}")

print(f"Additoin of {2:10} and {300:10} is = {2+3}")

Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

print(f"Additoin of {2:-10} and {'hello':10} is = {2+3}")

print(f"Additoin of {2:-10} and {'hello'!r} is = {2+3}}")

Other modifiers can be used to convert the value before it is formatted. '!a' applies ascii(), '!s' applies str(), and '!r' applies repr():

```
a="friends"
print(f"{a!r}")
```

output:

'friends'

# Formatted String Literals Python

print(f%(item)s %(qunt)d cost $% (price).2f' % {'item': 'Apple', 'qunt':23, 'price':200})

a=2

b=3

print(f'first={a} \nsecond={b}')

a=2

b=3

print(f'first={a:10d} \nsecond={b:10d}')

# format() string formatting method

print('We are the {} who say "{}".format('knights', 'Ni'))

I print('{0} and {1}'.format('spam', 'eggs'))

```
print('{1} and {0}'.format('spam', 'eggs'))

print('This {food} is {adjective}.'.format(food='spam',adjective='absolutely horrible'))

print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred', other='Georg'))

print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format({'Sjoerd': 4127, 'Jack': 4098, 'Dcab':8637678}))

print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**{'Sjoerd': 4127, 'Jack': 4098, 'Dcab':

8637678}))
```

# format() string formatting method

```
text = 'Deep'

print("\n")

# left aligned

print('{:<25}'.format(text)) # Right aligned

print("{:>25}'.format(text))

# centered

print('{:^25}'.format(text))
```

# format() string formatting method

```
number = 65

print("\n")

print("The number is:{:d}".format(number))

print('Output number in octal format: {0:0}'.format(number)) print('Output number in binary format: {0:b}'.format(number)) print('Output number in hexadecimal format:

{0:x}'.format(number))

print('Output number in HEXADECIMAL: {0:X}'.format(number))
```

# Python Statement

Python's statement set

Each statement in Python has its own

**syntax**-the rules that define its structure

**semantic**-specific meaning

Types of Python Statements

**Single Statements**

**Compound Statements**

**Comments**

# Python Single Statement

A simple statement is composed within a single logical line. Several simple statements may occur on a single line separated by semicolons

Semicolon(;) is optional.

Multiple Statements in Single Line

Single Statement in Multiple Line

# Python Single Statement

simple_stmt ::= expression_stmt

| assert_stmt

| assignment_stmt

| augmented_assignment_stmt

| annotated_assignment_stmt

|pass_stmt

| del_stmt

| return_stmt

| yield_stmt

| raise_stmt

# Python Compound Statement

Compound statements contain (groups of) other statements. All Python compound statements-statements that have other statements nested inside them-follow the same general pattern of a header line terminated in a colon, followed by a nested block of code usually indented underneath the header line, like this.

Header line: Nested statement block.

# Python Compound Statement Examples

compound_stmt ::=_if_stmt

| while_stmt

| for_stmt

| try_stmt

|with_stmt

| funcdef

| classdef

| async_with_stmt

| async_for_stmt

| async_funcdef

# What Python Removes

End-of-line is end of statement

Scope for compound statement

Parentheses are optional

# Comments

Comments are the useful information that the developers provide to make the reader understand the source code. Olt explains the logic or a part of it used in the code. There are two types of comment in Python:

**Single Line Comments**

**Multiline Comments**

# Single Line Comments

Python single line comment starts with hashtag symbol with no white spaces (#) and lasts till the end of the line. If the comment exceeds one line then put a hashtag on the next line and continue the comment. Python's single line comments are proved useful for supplying short explanations.

**Examples**

**1. sum = a + b # adding two integers**

**2. #Addition of two Number**

**A=2**

**B=3**

# Multiline Comments

Python_multi-line comment is a piece of text enclosed in a delimiter ("") on each end of the comment. Again there should be no white space between the delimiter (""""). They are useful when the comment text does not fit into one line; therefore needs to span across lines. Multi-line comments or paragraphs serve as documentation for others reading your code.

Example

“’’

This is

My Program

Of addition

“’’

# Ternary Operator in Python

Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false.

It was added to Python in version It simply allows to test a condition in a single line replacing the multiline if- else making the code compact.

# Syntax of Ternary Operator

[on_true] if [expression] else [on_false]

(if_test_is_false, if_test_is_true)[test]

{False:if_test_is_false,True:if_test_is_true}[test]

(lambda: if_test_is_false, lambda: if_test_is_true)[test]

**Note: Ternary operator can be written as nested if-else**

# Controll Statements

- Conditional
  - If else
- Looping
  - For loop
  - While loop
  - Iterator
- Jumping
  - Break
  - Continue
  - return

# Problem: 3 Integers

Statement:

You will be given 3 integers as input. The inputs may or may not be different · from each other. You have to output 1 if all three inputs are different from each other, and 0 if any input is repeated more than once. Input Three integers on three lines. Output 1 if the three inputs are different from each other, O if some input is repeated more than once.

| Test Case | Input | Output |
| --- | --- | --- |

| | | |
|---|---|---|
| Test 1 | 3<br>2<br>1 | 1 |
| Test 2 | 100<br>5<br>5 | 0 |

# Problem: Multiple

Statement:

You are given two integers, say M and N. You must check whether M is an exact multiple of N, without using loops. You have to output 0 if M is not a multiple of N. You have to output M/N if M is a multiple of N. Input Two integers, say M and N. Output You have to output 0 if M is not a multiple of N. You have to output M/N if M is a multiple of N.

| Test Case | Input | Output |
|---|---|---|
| Test 1 | 3<br>2 | 0 |
| Test 2 | 100<br>5 | 20 |

# Problem: Pythagorean

Statement:

triple of numbers (a,b,c) is called a Pythagorean triple if $a^2+b^2=c^2$. In this question, you will be given three numbers. You have to output 1 if the three numbers form a Pythagorean triple. Otherwise, you have to output 0. Note that the inputs may not be given in order: you may have to try all possible orderings of the three numbers to determine whether they form a Pythagorean triple. Input Three integers. Output 1 if the three numbers are part of a Pythagorean triple 0 otherwise.

| Test Case | Input | Output |
|---|---|---|
| **Test 1** | **3**<br>**5**<br>**4** | **1** |
| **Test 2** | **1**<br>**2**<br>**3** | **0** |

# Problem: ATM

Statement:

Pooja would like to withdraw X $US from an ATM. The cash machine will only accept the transaction if X is a multiple of 5, and Pooja's account balance has enough cash to perform the withdrawal transaction (including bank charges). For each successful withdrawal the bank charges 0.50 $US. Calculate Pooja's account balance after an attempted transaction.

**Input**

Positive integer 0 < x <= 2000 - the amount of cash which Pooja wishes to withdraw.
Nonnegative number 0<= Y <= 2000 with two digits of precision - Pooja's initial account balance.

**Output**

Output the account balance after the attempted transaction, given as a number with two digits

**Example - Successful Transaction**

Input:

30 120.00

Output:

89.50

**Example - Incorrect Withdrawal Amount (not multiple of 5)**

Input:

42 120.00

Output:

120.00

**Example - Insufficient Funds**

Input:

300 120.00

Output:

120.00

# Looping means

A loop statement allows us to execute a statement or group of statements multiple times.
Looping Statements

1. While Loop ( if programmer don't know how much time run loop then use)

2. For Loop ( if programmer know how much time run loop then use)

3. Iterator

# While Loop and Loop else

When combined with the loop else clause, the break statement can often be eliminated. The loop else clause is also run if the body of the loop is never executed

**General Format**

```
While condition:        #Loop test
        Statements      #Loop body
Else:                   #Optional else
        Statements      #Run if didn't exit loop with break
```

# For loop

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The for statement works on strings, lists, tuples, and other built- in iterables, as well as new user-defined objects.

**General Format**

```
for target in object:                    # Assign object items to target
statements                               # Repeated loop body: use target
else:                                    #Optional else part
statements                               # If we didn't hit a 'break'
```

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:
        print(a, b, c)
for (a, b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
        print(a, b, c)
```

# Problem: All Prime Number

Statement:

You are given an integer N. You need to print the series of all prime numbers till N.

**Input**

The first and only line of the input contains a single integer N denoting the number till where you need to find the series of prime numbers.

**Output**

Print the desired output in a single line separated by spaces.

| Test Case | Input | Output |
|---|---|---|
| Test 1 | 9 | 2 3 5 7 |

# Problem: Count Divisors

Statement:

You have been given 3 integers l, r and k. Find how many numbers between l and r (both inclusive) are divisible by k. You do not need to print these numbers, you just have to find their count.

**Input**

The first and only line of input contains 3 space separated integers I, r and k.

**Output**

Print the required answer on a single line.

| Test Case | Input | Output |
|---|---|---|
| Test 1 | 1 10 1 | 10 |

# Problem: Roy and Profile Picture

Statement:

Roy wants to change his profile picture on Facebook. Now Facebook has some restrictions over the dimensions of pictures that we can upload. Minimum dimension of the picture can be L x L, where L is the length of the side of the square.

Now Roy has N photos of various dimensions.

Dimension of a photo is denoted as W x H

where W - width of the photo and H - Height of the photo

When any photo is uploaded following events may occur:

[1] If any of the width or height is less than L, the user is prompted to upload another one. Print "UPLOAD ANOTHER" in this case.

[2] If width and height, both are large enough and

(a) if the photo is already square then it is accepted. Print "km" in this case.

(b) else user is prompted to crop it. Print "CROP IT" in this case.

(quotes are only for clarification)

Given L, N, W and H as input, print appropriate text as output.

**Input**

First line contains L.

Second line contains N, number of photos.

Following N lines each contains two space separated integers W and H.

**Output**

Print appropriate text for each photo in a new line.

| Test Case | Input | Output |
|---|---|---|
| Test 1 | 180<br>3<br>640 480<br>120 300<br>180 180 | <br><br>CROP IT<br>UPLOAD ANOTHER<br>ACCEPTED |

# Iterator Object

An iterator is an object that contains a countable number of values. An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods ____iter____() and ____next__().

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from. All these objects have an iter() method which is used to get an iterator.

# Collections/Sequence data type

There are two type of collections:

1. Unordered
    1. Set
    2. Frozenset
2. Ordered

1. Tuple
2. String
3. List
4. Dictionary

**Operators for collections**

| **Operators** | **Methods** | **Notes** |
|---|---|---|
| key in s | - | containment check |
| key not in s | - | non-cotainment check |
| s1==s2 | - | s1 is equivalent to s2 |
| s1!=s2 | - | s1 is not equivalent to s2 |
| s1<=s2 | Issubset() | s1 is subset of s2 ( every element of s1 must be in s2 and s2 maybe same as s1 like s1={1,2} s2{1,2} |
| s1<s2 | - | s1 is proper subset of s2 s2 ( every element of s1 must be in s2 and s2 not same as s1 like s1={1,2} s2{1,2,3} |
| s1>=s2 | Issuperset() | s1 is superset of s2(s1={1,2} s2{1,2}) s1 is a supber set |
| s1>s2 | - | s1 is proper superset of s2(s1={1,2,3} s2{1,2}) s1 is a supber set |
| s1\|s2 | union() | the union of s1 and s2 |
| s1&s2 | intersection() | the intersection of s1 and s2 (same values written) |
| s1-s2 | difference() | the set of elements in s1 but not s2 |

| | | |
|---|---|---|
| s1^s2 | symmetric_difference() | the set of elements in precisely one of s1 or s2(remove same values from both set and give unique values from both sets) |

**Built-in Functions for collections**

1. **all()**
2. **any()**
3. **enumerate()**
4. **len()**
5. **max()**
6. **min()**
7. **sorted()**
8. **sum()**

**all () Method**

If the iterable is empty, return True.Return True if bool(x) is True for all values x in the iterable. Give False `val={False,0,0.0,0.00,""} if any of them values are used`

| | |
|---|---|
| Parameters | (iterable: Iterable[object], /) |
| Return | bool |

**any () Method**

If the iterable is empty, return False.Return True if bool(x) is True for any x in the iterable.

| | |
|---|---|
| Parameters | (iterable: Iterable[object], /) |
| Return | bool |

**enumerate () Method**

**iterable:** an object supporting iteration

The enumerate object yields pairs containing a count (from start, which defaults to zero) and a value yielded by the iterable argument. enumerate is useful for obtaining an indexed list: (0, seq[0]), (1, seq[1]), (2, seq[2]), ... Return an enumerate object.

| | |
|---|---|
| Parameters | (iterable: Iterable[_T], start: int=...) |
| Return | Enumrate object |

**len () Method**

Return the number of items in a container. `val={1,True,1.00} these value is 1 and lenth is 1 in a set. val={False,0,0.0,0.00} these value is 0 and lenth is 1 in a set.`

| | |
|---|---|
| Parameters | (obj: Sized) |
| Return | int |

**max () Method**

a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty. With two or more arguments, return the largest argument.

```
val={0,0.0,0.00,False}
print(max(val))
output is 0 it will take 1st element from collection if all values are same.

val={True,1}
print(max(val))
output is True it will take 1st element from collection if all values are same.
```

| | |
|---|---|
| Parameters | Iterable object |
| Return | Value depend on type |

**min () Method**

With a single iterable argument, return its smallest item. The default keyword-only argument specifies an object to return if the provided iterable is empty. With two or more arguments, return the smallest argument.

| | |
|---|---|
| Parameters | Iterable object |
| Return | Value depend on type |

**sorted () Method**

A custom key function can be supplied to customise the sort order, and the reverse flag can be set to request the result in descending order. Return a new list containing all items from the iterable in ascending order.

| | |
|---|---|
| Parameters | (iterable: Iterable[SupportsLessThanT], /, *, key: None=..., reverse: bool=...)<br><br>(iterable: Iterable[_T], /, *, key: Callable[[_T], SupportsLessThan], reverse: bool=...) |
| Return | List[SupportsLessThanT]<br><br>List[_T] |

**sum () Method**

**This function is intended specifically for use with numeric values and may reject non-numeric types. Return the sum of a 'start' value (default: 0) plus an iterable of numbers. When the iterable is empty, return the start value.**

| | |
|---|---|
| Parameters | (iterable: Iterable[_T], /)<br><br>(iterable: Iterable[_T], /, start: _S) |
| Return | Union[_T, int]<br><br>Union[_T, _S] |

# Unordered Sequence

Unordered sequences include set and frozenset.

## Set

A set is a collection, which is unordered, unindexed ,mutable, iterable and does not allow duplicate values. Sets are written with { } curly bracket.

**Unordered:** means that the items in a set do not have a defined order.

**Unindexed:** means that Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

**Mutable:** means that we can change the items after the set has been created. Once a set is created, you can change its items, you can add new items or remove old items.

**Iterable:** means that get values one by one using loop or iterator.

**No duplicates values:** means that Sets cannot have two items with the same value.

**Notes:**

We cannot create sets as a subset or multiple sets as subsets. We can create a frozenset as a subset or multiple frozensets in a set.

### The set() Constructor or method

set() constructor or method used to make a set. It is also used to make empty sets. The syntax of set function is

**set(iterable) or {iterable}**

### Built-in methods in set

Set built-in methods are given below

1. **add()**
2. **remove()**
3. **discard()**
4. **clear()**
5. **copy()**
6. **difference()**
7. **difference_update()**
8. **symmetric_difference()**
9. **symmetric_difference_update()**
10. **Union()**
11. **update()**
12. **intersection()**
13. **intersection_update()**
14. **isdisjoint()**
15. **issubset()**

16. **issuperset()**
17. **pop()**

**add() Methods in Set**

The set add() method adds a given element to a set. If the element is already present, it doesn't add any element.

| | |
|---|---|
| Parameters | single element or tuple or frozenset with single string |
| Return | None |

**remove() Methods in Set**

The remove() removes the specified element from the set and updates the set. It doesn't return any value. If the element is not a member, it raises a KeyError.

| | |
|---|---|
| Parameters | Single Element or tuple or frozenset with single string |
| Return | None |

**discard() Methods in Set**

The discard() method removes a specified element from the set (if present). If the element is not a member, it does not give any error.

| | |
|---|---|
| Parameters | Single Element or tuple or frozenset with single string |
| Return | None |

**clear() Methods in Set**

The clear() method removes all elements from the set.

| | |
|---|---|
| Parameters | None |
| Return | None |

**copy() Methods in Set**

The copy() method returns a shallow copy of the set

| | |
|---|---|
| Parameters | None |
| Return | Shallow copy of Set |

**difference() Methods in Set**

The difference() method returns the set difference of two sets. difference() method returns the difference between two sets which is also a set. It doesn't modify original sets.(like: A-B).

| | |
|---|---|
| Parameters | Iterable Object |
| Return | set |

**Difference_update() Methods in Set**

The method returns the set difference of two sets. The method returns the difference between two sets which is also a set. It does modify original sets.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | None |

**symmetric_difference() Methods in Set**

The symmetric difference of two sets A and B is the set of elements that are in either A or B, but not in their intersection. But not make any changes in the original set.(like: A-B and B-A or remove common elements from two sets).

| | |
|---|---|
| Parameters | Iterable Object |
| Return | set |

**symmetric_difference_update() Methods in Set**

The symmetric difference update of two sets A and B is the set of elements that are in either A or B, but not in their intersection and make any changes in the original set.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | None |

**Union() Methods in Set**

The union of two or more sets is the set of all distinct elements present in all the sets..

| | |
|---|---|
| Parameters | Iterable Objects |
| Return | set |

**update() Methods in Set**

The Python set update() method updates the set, adding items from other iterables.

| | |
|---|---|
| Parameters | Iterable Objects separated by comma |
| Return | None |

**Intersection() Methods in Set**

The intersection() method returns a new set with elements that are common to all sets.It doesn't modify original sets.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | set |

**Intersection_update() Methods in Set**

The intersection() method finds common elements in all sets. It does modify original sets.

| | |
|---|---|
| Parameters | Iterable Object |
| dxReturn | None |

**isdisjoint() Methods in Set**

isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False. Two sets are said to be disjoint sets if they have no common elements.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | Boolean |

**issubset() Methods in Set**

The issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | Boolean |

**issuperset() Methods in Set**

The issuperset() method returns True if all elements of another set are present in it.If not, it returns False.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | Boolean |

**pop() Methods in Set**

The pop() method removes an arbitrary element from the set and returns the elements removed. If the set is empty, a TypeError exception is raised.

| | |
|---|---|
| Parameters | None |
| Return | Element |

**Use of Set**

1. To perform mathematical operation
2. To remove duplicate from list
3. Order matters in sequences but not in set

4. Database query

## Frozen Set

A frozenset is a collection, which is unordered, unindexed ,immutable, iterable and does not allow duplicate values. frozenset Sets written frozenset({ }).

**Unordered:** means that the items in a set do not have a defined order.

**Unindexed:** means that Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

**immutable:** means that we can not change the items after the set has been created.

**Iterable:** means that get values one by one using loop or iterator.

**No duplicates values:** means that Sets cannot have two items with the same value.

Note:

We can create a frozenset as a subset or multiple forzenset as subsets. We cannot create sets as a subfrozenset or multiple sets as subfrozensets.

### The frozenset() Constructor or method

frozenset() constructor or method used to make a frozenset. It is also use to make empty frozenset.The syntax of frozenset function is

**frozenset(iterable)**

### Methods in frozenset

Like normal sets, frozenset can also perform different operations like

1. copy()
2. difference()
3. symmetric_difference()
4. union()
5. intersection()
6. isdisjoint()
7. issubset()
8. issuperset()

#### copy() Methods in frozenSet

The copy() method returns a shallow copy of the set

| | |
|---|---|
| Parameters | None |
| Return | Shallow copy of frozenSet |

**difference() Methods in frozenSet**

The difference() method returns the set difference of two sets. difference() method returns the difference between two sets which is also a set. It doesn't modify original sets.(like: A-B).

| | |
|---|---|
| Parameters | Iterable Object |
| Return | frozenSet |

**symmetric_difference() Methods in frozenSet**

The symmetric difference of two sets A and B is the set of elements that are in either A or B, but not in their intersection. But not make any changes in the original set.(like: A-B and B-A or remove common elements from two sets).

| | |
|---|---|
| Parameters | Iterable Object |
| Return | frozenSet |

**Union() Methods in frozenSet**

The union of two or more sets is the set of all distinct elements present in all the sets..

| | |
|---|---|
| Parameters | Iterable Objects |
| Return | frozenSet |

**Intersection() Methods in frozenSet**

The intersection() method returns a new set with elements that are common to all sets.It doesn’t modify original sets.

| | |
|---|---|
| Parameters | Iterable Object |

| | |
|---|---|
| Return | frozenSet |

**isdisjoint() Methods in frozenSet**

isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False. Two sets are said to be disjoint sets if they have no common elements.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | Boolean |

**issubset() Methods in frozenSet**

The issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | Boolean |

**issuperset() Methods in frozenSet**

The issuperset() method returns True if all elements of another set are present in it.If not, it returns False.

| | |
|---|---|
| Parameters | Iterable Object |
| Return | Boolean |

**Use of frozenset**

1. Key in dictionary
2. Element in set

# Ordered Sequence

Ordered sequences include Tuple, string, list and dictionary.

**Indexing**

tupleReference[index]

**Slicing**

1. tupleReference[:]
2. tupleReference[::]
3. tupleReference[][]

## Tuple

A tuple is a collection, which is ordered, indexed ,immutable, iterable and allows duplicate values. Tuple written with ( ) round brackets.

### Creating Tuple

Empty Tuple (). Tuple with one element (5,). The tuple() Constructor or method

### Operations on Tuple

1. Change mutable item in tuple – add list in tuple and do this.
2. Concatenation using +
3. Multiplying using *

### Methods in Tuple

1. count()
2. index()

#### count() Method in tuple

count() will return the number of times and element appears in an tuple

| | |
|---|---|
| Parameters | Single element or iterable object |
| Return | int |

#### index() Method in tuple

index(value,start_index,stop_index) returns the first index of the matching value. If the index value is not found it will give ValueError.

| | |
|---|---|
| Parameters | Single element or iterable object |
| Return | int |

**use of tuple**

1. Tuple are faster
2. Tuple uses to create constants
3. Tuple are use to create dictionary

## String Literals

Collection of characters in single quotes(‘ ’) or double quotes(‘ ’) or triple quotes(‘ ’) is called string literals. A string is a collection, which is ordered , indexed, immutable, iterable and allows duplicate values. Nested Structure is not allowed.

Three types of String Literals:

1. Simple String
2. Format String ( use F or f for formatting string)
3. Row String ( use R or r for raw string)

**print()** used to print string.

**input()** used to take string from the user.

**Operations on string**

1. Concatenation using +
2. Multiplying using *
3. Membership operators

**String Conversion tools**

“42”+1
TypeError: Can’t conert ‘int’ object to str implicity
**str() method**
**reper() method**

1. Concatenation using +
2. Multiplying using *
3. Membership operators

**Creating string**

We can create a string using single quotes( ‘ ‘ ) or double quotes( “ “ ) or single triple quotes( ‘’’ ‘’’) or double triple quotes( “”” “””). str() Constructor or method. repr() Constructor or method.repr() method used to show string in string form like: True into ‘true’.

**Character String Methods ( work on single character)**

1. capitalize()
2. casefold()
3. lower()
4. islower()
5. upper()
6. isupper()
7. isalpha()
8. isalnum()
9. swapcase()
10. isdigit()
11. isnumeric()
12. isdecimal()

**capitalize () Method in string**

capitalize() method Capitalise a first character of a string.

| | |
|---|---|
| Parameters | None |
| Return | string |

**casefold () Method in string**

casefold() method converts every word into lower case like: People to people. This is aggressive.

| | |
|---|---|
| Parameters | None |
| Return | string |

**lower () Method in string**

lower() method converts every word into lower case like: People to people. This is not aggressive.

| | |
|---|---|
| Parameters | None |
| Return | string |

**islower () Method in string**

islower() method if all characters in lower case return true otherwise false.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**upper () Method in string**

upper() method converts every word into Upper case like: People to PEOPLE.

| | |
|---|---|
| Parameters | None |
| Return | string |

**isupper () Method in string**

isupper() method if all characters in Upper case return true otherwise false.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**isalpha () Method in string**

isalpha() method if all the characters in string are alphabets return true otherwise false.if space then also false.

| | |
|---|---|
| Parameters | None |

| | |
|---|---|
| Return | boolean |

**isalnum () Method in string**

isalnum() method if number and alphabets and both return true otherwise false.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**swapcase () Method in string**

swapcase() method converts upper case not lower case and lower case into upper case.

| | |
|---|---|
| Parameters | None |
| Return | string |

**isdigit () Method in string**

isdigit() method returns true decimal ( 0-9: like:” 3”) otherwise false.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**isnumeric () Method in string**

isnumeric() return true decimal (0-9: like:” 3”)) and subscript(like: “$3_2$”) and superscript(like: “$3^2$”)

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**isdecimal () Method in string**

isdecimal() return true decimal (0-9: like:” 3”) and subscript(like: “$3_2$”) and superscript(like: “$3^2$”) and fractional part(like: “3 ½”) otherwise false.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**String Formatting Methods**

1. ljust(width,fill)
2. rjust(width,fill)
3. center(width, fill)
4. expandtabs(tabsize)
5. format(fmtstr, *args, **kwargs)

**ljust(width,fill) Method in string**

ljust(width,fill) method means left space justifying. Padding is done using the specified fill character (default is a space). Return a left-justified string of length width.

| | |
|---|---|
| Parameters | ljust(width: int, fillchar:str) |
| Return | string |

**rjust (width, fill) Method in string**

rjust(width,fill) method means right space justifying. Padding is done using the specified fill character (default is a space). Return a right-justified string of length width.

| | |
|---|---|
| Parameters | rjust(width: int, fillchar:str) |
| Return | string |

**center(width,fill) Method in string**

center(width,fill) method means centre space justifying. Padding is done using the specified fill character (default is a space). Return a centre string of length width.

| | |
|---|---|
| Parameters | rjust(width: int, fillchar:str) |
| Return | string |

**expandtabs(tabsize) Method in string**

justify tab size. If tab size is not given, a tab size of 8 characters is assumed. Return a copy where all tab characters are expanded using spaces.

| | |
|---|---|
| Parameters | (tabsize: int) |
| Return | string |

**format(fmtstr, *args, **kwargs) Method in string**

The substitutions are identified by braces ('{' and '}'). Return a formatted version of S, using substitutions from args and kwargs.

| | |
|---|---|
| Parameters | (*args: object, **kwargs: object) |
| Return | string |

**String Manipulate Methods**

1. isidentifier()
2. isprintable()
3. isspace()
4. istitle()
5. Istrip(chars)
6. rstrip(chars)
7. strip(chars)
8. partition(sep)
9. rpartition(sep)
10. rsplit([sep[, maxsplit]])
11. split([sep [,maxsplit]])
12. splitlines([keepends])

**isidentifier () Method in string**

Call keyword.iskeyword(s) to test whether string s is a reserved identifier,
such as "def" or "class". Return True if the string is a valid Python identifier, False otherwise.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**isprintable () Method in string**

A string is printable if all of its characters are considered printable in repr() or if it is empty. Return True if the string is printable, False otherwise.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**isspace () Method in string**

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string. Return True if the string is a whitespace string, False otherwise.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**istitle () Method in string**

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones. Return True if the string is a title-cased string, False otherwise.

| | |
|---|---|
| Parameters | None |
| Return | boolean |

**lstrip (chars) Method in string**

If chars is given and not None, remove characters in chars instead. Return a copy of the string with leading whitespace removed.

| | |
|---|---|
| Parameters | chars |
| Return | string |

**rstrip (chars) Method in string**

If chars is given and not None, remove characters in chars instead. Return a copy of the string with trailing whitespace removed.

| | |
|---|---|
| Parameters | chars |
| Return | string |

**strip (chars) Method in string**

If chars is given and not None, remove characters in chars instead. Return a copy of the string with leading and trailing whitespace removed.

| | |
|---|---|
| Parameters | chars |
| Return | string |

**partition (sep) Method in string**

Partition the string into three parts using the given separator. This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it. If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

| | |
|---|---|
| Parameters | sep |
| Return | Tuple[str, str, str] |

**rpartition (sep) Method in string**

Partition the string into three parts using the given separator. This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it. If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

| | |
|---|---|
| Parameters | sep |
| Return | Tuple[str, str, str] |

**rsplit ([sep[, maxsplit]]) Method in string**

**Sep**

The separator used to split the string. When set to None (the default value), will split on any whitespace character (including \\n \\r \\t \\f and spaces) and will discard empty strings from the result.

**Maxsplit**

Maximum number of splits (starting from the left). -1 (the default value) means no limit. Splitting starts at the end of the string and works to the front.

Return a list of the substrings in the string, using sep as the separator string.

| | |
|---|---|
| Parameters | ([sep[, maxsplit]]) |
| Return | List[str] |

**split ([sep[, maxsplit]]) Method in string**

**sep**
The separator used to split the string. When set to None (the default value), will split on any whitespace
character (including \\n \\r \\t \\f and spaces) and will discard
empty strings from the result.
**maxsplit**
Maximum number of splits (starting from the left).-1 (the default value) means no limit.
**Note**: str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

Return a list of the substrings in the string, using sep as the separator string.

| | |
|---|---|
| Parameters | ([sep[, maxsplit]]) |
| Return | List[str] |

**splitlines ([keepends]) Method in string**

Line breaks are not included in the resulting list unless keepends are given and true. Return a list of the lines in the string, breaking at line boundaries.

| | |
|---|---|
| Parameters | ([keepends]) |
| Return | List[str] |

**Sub-String Methods**

1. count(sub, start, end)
2. startsWith(prefix/tuple, start, end)
3. endsWith(suffix/tuple, start, end)
4. rfind(sub,start,end)
5. find(sub, start, end)
6. rindex(sub, start, end)
7. index(sub, start, end)
8. replace(old, new, count)

**count(sub, start, end) Method in string**

Optional arguments start and end are interpreted as in slice notation. Return the number of non-overlapping occurrences of substring sub in string S[start:end].

| | |
|---|---|
| Parameters | (sub, start, end) |
| Return | int |

**startsWith(prefix/tuple, start, end) Method in string**

With optional start, test S beginning at that position.With optional end, stop comparing S at that position.prefix can also be a tuple of strings to try. Return True if S starts with the specified prefix, False otherwise.

| | |
|---|---|
| Parameters | (prefix/tuple, start, end) |
| Return | boolean |

**endsWith(suffix/tuple, start, end) Method in string**

With optional start, test S begins at that position. With optional end, stop comparing S at that position.suffix can also be a tuple of strings to try. Return True if S ends with the specified suffix, False otherwise.

| | |
|---|---|
| Parameters | (suffix/tuple, start, end) |
| Return | boolean |

**rfind(sub,start,end) Method in string**

such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure. Return the highest index in S where substring sub is found,

| | |
|---|---|
| Parameters | (sub,start,end) |
| Return | int |

**find(sub, start, end) Method in string**

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure. Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

| | |
|---|---|
| Parameters | (sub, start, end) |
| Return | int |

**rindex(sub, start, end) Method in string**

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found. Return the highest index in S where substring sub is found,such that sub is contained within S[start:end].

| | |
|---|---|
| Parameters | (sub, start, end) |
| Return | int |

**index(sub, start, end) Method in string**

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found. Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

| | |
|---|---|
| Parameters | (sub, start, end) |
| Return | int |

**replace(old, new, count) Method in string**

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.If the optional argument count is given, only the first count occurrences are replaced.Return a copy with all occurrences of substring old replaced by new.

| | |
|---|---|
| Parameters | (old, new, count) |
| Return | string |

**Encoding String Methods**

1. S.encode(encoding,errors)
2. S.zfill(width)
3. S.join(iterable)
4. S.maketrans(x[, y[, z]])
5. S.translate(map)

**encode(encoding,errors) Method in string**

Encode the string using the codec registered for encoding.
**encoding**
The encoding in which to encode the string.
**errors**
The error handling scheme to use for encoding errors.
The default is 'strict' meaning that encoding errors raise aUnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs. register_error that can handle UnicodeEncodeErrors.

| | |
|---|---|
| Parameters | (encoding,errors) |
| Return | bytes |

**zfill(width) Method in string**

Pad a numeric string with zeros on the left, to fill a field of the given width. The string is never truncated.

| | |
|---|---|
| Parameters | (width: int) |
| Return | string |

**join(iterable) Method in string**

Concatenate any number of strings. The string whose method is called is inserted in between each given string. The result is returned as a new string. Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

| | |
|---|---|
| Parameters | (iterable: Iterable[str]) |
| Return | string |

**maketrans(x[, y[, z]]) Method in string**

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result. Return a translation table usable for str.translate().

| | |
|---|---|
| Parameters | (x: Union[Dict[int, _T], Dict[str, _T], Dict[Union[str, int], _T]], /) -> Dict[int, _T]<br>def maketrans(x: str, y: str, z: Optional[str]) |
| Return | Dict[int, Union[int, None]] |

**translate(map) Method in string**

Replace each character in the string using the given translation table.

**table**

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list.

If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

| | |
|---|---|
| Parameters | (table: Union[Mapping[int, Union[int, str, None]], Sequence[Union[int, str, None]]]) |
| Return | string |

**Character code conversions methods**

1. ord()
2. chr()
3. bin()

**ord() Method in string**

ord() method converts a single character to its underlying integer code. Return the Unicode code point for a one-character string.

| | |
|---|---|
| Parameters | Single character in form of string |
| Return | int |

**chr() Method in string**

chr() method taking an integer code and converting it to the corresponding character. Return a Unicode string of one character with ordinal i; 0 <= i <= 0x10ffff.

| | |
|---|---|
| Parameters | int |
| Return | string |

**bin() Method in string**

bin() method converts integer into binary. Return the binary representation of an integer in string form.

| | |
|---|---|
| Parameters | int |
| Return | string |

**Basic Operations on String**

1. String Concatenations +
2. String Repetition *
3. Membership Operator ( in,not in )

**use of string literals**

## List

list is ordered , indexed, mutable, iterable and allows duplicate values. Lists are Python's most flexible ordered collection object type. Lists are dynamic.

**Creation a list**

Empty list [ ]. The list() Constructor or method

**Methods on list**

1. append()
2. extend()
3. insert()
4. index()
5. count()
6. sort()
7. reverse()
8. copy()
9. clear()
10. pop()
11. remove()

**append () Method in string**

Append objects to the end of the list.

| | |
|---|---|
| Parameters | (object: _T) |

| | |
|---|---|
| Return | None |

**extend () Method in string**

Extend the list by appending elements from the iterable.

| | |
|---|---|
| Parameters | (iterable: Iterable[_T]) |
| Return | None |

**insert () Method in string**

Insert object before index.

| | |
|---|---|
| Parameters | (index: int, object: _T) |
| Return | None |

**index () Method in string**

Raises ValueError if the value is not present. Return the first index of value.

| | |
|---|---|
| Parameters | value: _T, start: int=..., stop: int=...) |
| Return | int |

**count () Method in string**

Return number of occurrences of value.

| | |
|---|---|
| Parameters | (value: _T) |
| Return | int |

**sort () Method in string**

sort the list in ascending order and return None. The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained). If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values. The reverse flag can be set to sort in descending order.

| | |
|---|---|
| Parameters | (*, key: None=..., reverse: bool=...)<br><br>(*, key: Callable[[_T], SupportsLessThan], reverse: bool=...) |
| Return | None |

**reverse () Method in string**

Reverse *IN PLACE*.

| | |
|---|---|
| Parameters | None |
| Return | None |

**copy () Method in string**

Return a shallow copy of the list.

| | |
|---|---|
| Parameters | None |
| Return | List[_T] |

**clear() Method in string**

Remove all items from the list.

| | |
|---|---|
| Parameters | None |
| Return | None |

**pop() Method in string**

Raises IndexError if list is empty or index is out of range. Remove and return item at index (default last).

| | |
|---|---|
| Parameters | (index: int=..., /) |
| Return | _T |

**remove() Method in string**

Remove first occurrence of value.Raises ValueError if the value is not present..

| | |
|---|---|
| Parameters | (value: _T, /) |
| Return | None |

**Operations on List**

1. Concatenation using +
2. Multiplying using *
3. Membership operators in list

**Use of list**

lists are so flexible, you're probably best off using them rather than tuples the majority of the time.

The simplest ways to represent matrices (multidimensional arrays) in Python is as lists with nested sublists.s

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

## Dictionary

A collection that allows us to look up information associated with keys is called a mapping. Python dictionaries are mappings. Some other programming languages provide similar structures called hshes or associative arrays. A dictionary can be created in Python by listing key-value pairs inside of curly braces.

Example: D={‘‘name’ : ”bob”,age’:40}.Notice that keys and values are joined with a “:” and commas are used to separate the pairs.

A dictionary is an example of a key value store also known as Mapping in Python. It allows you to store and retrieve elements by referencing a key. As dictionaries are referenced by key, they have very fast lookups. As they are primarily used for referencing items by key, they are not sorted.

**Why dictionary**

Lists allow us to store and retrieve items from sequential collections. When we want to access an item in the collection, we look it up by index-its position in the collection. Many applications require a more flexible way to look up information. For example, we might want to retrive information about students or employees based on their ID numbers.

**Note:**

Dictionary is written with curly brackets. Mappings are inherently unordered. When a dictionary is printed out, the order of keys will look essentially random. If you want to keep a collection of items in a certain order, you need a sequence, not a mapping. Variable-length, heterogeneous, and arbitrarily nestable. Of the category “mutable mapping".general, keys can be any immutable type, and values can be any type at all. dictionaries do not allow duplicate keys.

**Create a dictionary dict()**

1. You can also call the built-in type dict to create a dictionary in a way that, while usually less concise, can sometimes be more readable.

2. dict(x=42, y=3.14, z=7) # Dictionary with three items, str keys

3. dict([(1, 2), (3, 4)]) # Dictionary with two items, int keys

4. dict([(1,'za'), ('br',23)]) # Dictionary with mixed key types

5. dict() # Empty dictionary

6. dict(zip(keyslist, valueslist))

7. dict.fromkeys(['name', 'age'])

8. You can also create a dictionary by calling dict.fromkeys. The first argument is an iterable whose items become the keys of the dictionary; the second argument is the value that corresponds to each and every key (all keys initially map to the same value). If you use the second argument, it defaults to None. For example:

9. dict.fromkeys('hello', 2) # same as {'h':2, 'e':2, 'l':2, 'o':2}

10. dict.fromkeys([1, 2, 3]) # same as {1:None, 2:None, 3:None}

**Methods on dictionary**

1. keys()

2. values()

3. items()

4. copy()

5. clear()

6. update()

7. get()

8. pop()

9. setdefault()

10. popitem()

11. fromkeys()

**keys () Method in string**

a set-like object providing a view on D's keys.

| | |
|---|---|
| Parameters | None |
| Return | KeysView[_KT] |

**values () Method in string**

An object providing a view on D's value.

| | |
|---|---|
| Parameters | None |
| Return | ValuesView[_VT] |

**items () Method in string**

A set-like object providing a view on D's items.

| | |
|---|---|
| Parameters | None |
| Return | ItemsView[_KT, _VT] |

**copy () Method in string**

a shallow copy of D.

| | |
|---|---|
| Parameters | None |
| Return | Dict[_KT, _VT] |

**clear () Method in string**

Remove all items from D.

| | |
|---|---|
| Parameters | None |
| Return | None |

**update () Method in string**

D.update([E, ]**F) -> None. Update D from dict/iterable E and F. If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v
In either case, this is followed by: for k in F: D[k] = F[k]

| | |
|---|---|
| Parameters | (m: Mapping[_KT, _VT], /, **kwargs: _VT)<br>(m: Iterable[Tuple[_KT, _VT]], /, **kwargs: _VT)<br>(**kwargs: _VT) |
| Return | None |

**get () Method in string**

| | |
|---|---|
| Parameters | (key: _KT)<br>(key: _KT, default: Union[_VT_co, _T]) |
| Return | Optional[_VT_co]<br>Union[_VT_co, _T] |

**pop () Method in string**

| | |
|---|---|
| Parameters | (key: _KT)<br>(key: _KT, default: Union[_VT, _T]=...) |

| | |
|---|---|
| Return | _VT<br>Union[_VT, _T] |

**setdefault () Method in string**

Insert key with a value of default if key is not in the dictionary. Return the value for key if key is in the dictionary, else default.

| | |
|---|---|
| Parameters | (key: _KT, default: _VT=..., /) |
| Return | _VT |

**popitem () Method in string**

Remove and return a (key, value) pair as a 2-tuple. Pairs are returned in LIFO (last-in, first-out) order.
Raises KeyError if the dict is empty.

| | |
|---|---|
| Parameters | None |
| Return | Tuple[_KT, _VT] |

**fromkeys () Method in string**

Create a new dictionary with keys from iterable and values set to value.

| | |
|---|---|
| Parameters | (iterable: Iterable[_T], /)<br>(iterable: Iterable[_T], value: _S, /) |
| Return | Dict[_T, Any]<br>Dict[_T, _S] |

**Basic Operations on dictionary**

Membership operators in dictionary(k in D)

Indexing a Dictionary

Deleting dictionary

Finding length

Checking type