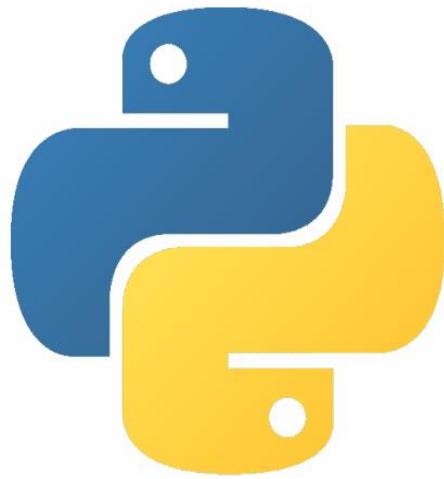


For B.E. B.C.A. M.C.A. M.C.M. B.S.C. Polythenic



# Python

by Aditya Choudhari Sir



**CCIT**  
Keeping Pace with Technology

An ISO 9001 : 2008 Certified Company

website: [www.ccitindia.com](http://www.ccitindia.com)

Rajapeth: 0721-2563615

## Table of Contents

### Introduction

Python Features .....	11
Python Installation .....	12
Python IDEs and Code Editors .....	14
General Format .....	17
print Function.....	18

### Variables and Data Types

Variables.....	20
Data Types.....	20
Identifiers .....	21
Keywords.....	21
Numbers Datatype .....	22
String Datatype .....	23
Boolean Datatype.....	23

### Operators

Operator Type .....	25
Multiple Statements .....	30
Multi-Line Statements.....	30
Multiple Assignments.....	31
Comments .....	31
Type Casting .....	32
Input Function.....	33

### Condition and loop Structure

if...else Statements.....	38
if...elif...else Statements .....	40
Ternary Operator .....	42
while Loop .....	43
while...else Loop.....	45
for Loop .....	46
range function.....	48
for...else Loop.....	51
break Statement .....	52
continue Statement .....	53

pass Statement.....	54
---------------------	----

## Functions

Function introduction .....	56
Function with Arguments.....	57
Function with Default Arguments.....	59
Function returning value.....	60
Function Arbitrary Arguments .....	62
Function Keyword Arguments .....	63
Recursive Function.....	64
Function Aliases / References .....	65
Nested Functions .....	66
Functions as Arguments.....	67
Functions Returning Functions .....	68
Function Decorator .....	69
Anonymous Functions.....	71
Global Variable .....	72
Local and Nonlocal Variable .....	74
Closures .....	76

## Object-Oriented Programming

Introduction .....	79
Class .....	80
Objects .....	81
Constructors.....	83
Destructor .....	86
Class Variables.....	87
Class methods .....	90
Static Methods .....	92
Passing Object as Argument .....	94
Special Method .....	96
Operator Overloading .....	97
Inheritance .....	105
Inheritance Method Overriding .....	108
Super( ) function .....	113
Access Modifiers .....	121
property function.....	124

property decorators .....	126
Polymorphism .....	129
Abstract Methods and Classes .....	134
Python Abstract Classes .....	134

## Collection

String .....	139
bytes Objects.....	152
bytearray Object .....	153
Lists.....	154
Tuple.....	160
Set .....	164
Frozen sets .....	169
Dictionary .....	172
map function.....	178
Array.....	180

## Modules

Module Introduction.....	184
Built-in Modules.....	184
Import Statement.....	185
from import Statement.....	185
Renaming a module .....	186
User defined Modules.....	186
Packages.....	190
Packages init.....	191
Subpackages.....	192
Main Functions in Python .....	194

## Exception Handling

Exception Introduction .....	196
Built-in Exceptions.....	197
try and except .....	198
try and except...else block.....	200
try and except...finally block .....	201
raise statement .....	202
custom Exception.....	203

**Iterators**

Iterators Introduction .....	204
Iterable class .....	207
StopIteration Exception .....	208
Generators .....	211
Yield statement .....	211
Itertools module.....	213

**Regular Expressions**

Regular Expressions Introduction .....	217
Meta Characters.....	225
Re module .....	226
Flags .....	227
Match object .....	228

**Database**

Database Introduction .....	230
SQLite Connection.....	233
SQLite Create table .....	235
SQLite insert,update,delete .....	236
SQLite Retrieve records.....	238
MySQL Connection.....	240
MySQL Create table .....	243
MySQL insert,update,delete .....	244
MySQL Retrieve records.....	248
Parameterized statement .....	250

**Datetime**

Datetime Introduction .....	252
Date class .....	253
Time class .....	257
Datetime class.....	259
Format Codes .....	263
Timedelta class.....	265
Calendar Module.....	268
time Module.....	271

## File I/O

File I/O Introduction.....	273
File Open .....	275
with statement.....	276
File read.....	278
File write.....	279
Serialization.....	281
Pickle module.....	281
Binary File read .....	282
Binary File write .....	283
os module.....	285
os.path module .....	289

## MultiThreading

Multithreading Introduction .....	291
Threading module.....	292
Thread class.....	293
Daemon Thread .....	300
Thread Synchronization .....	301
Lock class.....	301
Deadlock.....	303

## Multiprocessing

Multiprocessing Introduction .....	306
Multiprocessing module .....	307
Process class.....	307
Daemon Process.....	310
Process Synchronization .....	311
Lock class.....	311
Multiprocessing shared memory .....	312
class Value.....	313
class Array .....	314
Managers .....	315
SyncManager.....	315
BaseManager .....	317
Process Pools.....	319

**Subprocess**

subprocess	Introduction .....	321
Popen	class .....	322
Subprocess	Function .....	325
CompletedProcess	.....	326

**Tkinter**

Tkinter	Introduction .....	329
Tkinter	module .....	330
Tk	class .....	330
mainloop	.....	332
Button	.....	333
Entry	.....	336
Variable classes	.....	339
Label	.....	341
Layout management	.....	343
Pack manager	.....	343
Grid manager	.....	346
Place manager	.....	348
spinbox	.....	350
scale	.....	352
Checkboxes	.....	354
Radio Buttons	.....	357
Listbox	.....	360
Canvas	.....	364
Message Box	.....	369
simpdialog	.....	371
File Chooser	.....	372
Color Chooser	.....	375
PhotoImage	.....	376
Menus	.....	379
Text	.....	382
Scrollbar	.....	384
Frame	.....	386
Label Frame	.....	388
Events and Binds	.....	390

## Networking

Networking Introduction .....	395
Internet Protocol .....	396
Sockets .....	397
Class socket .....	398
Server socket .....	399
Client socket .....	400

## sys Module

sys Introduction .....	401
Command Line Arguments .....	403
stdin, stdout, stderr .....	404

## math Module

math Introduction .....	405
Constant .....	406
Number-theoretic and representation .....	406
Angular conversion .....	406
Power and logarithmic .....	407
Trigonometric .....	407

## random Module

random Introduction .....	408
Functions for integers .....	409
Functions for sequences .....	409
Real-valued distributions .....	410

## Built-in Functions

# Introduction

# Python

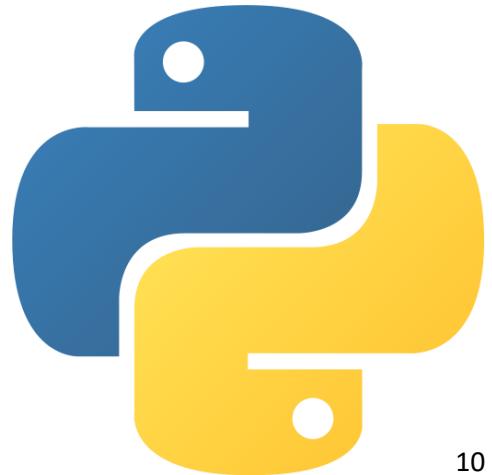
Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms. Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where other languages use punctuation, and it has fewer syntactical constructions than other languages.

Python is an object-oriented programming language created by Guido Rossum in 1989. It is ideally designed for rapid prototyping of complex applications. It has interfaces to many OS system calls and libraries and is extensible to C or C++. Many large companies use the Python programming language include NASA, Google, YouTube, etc. Python programming is widely used in Artificial Intelligence, Natural Language Generation, Neural Networks and other advanced fields of Computer Science. Python had deep focus on code readability & this class will teach you python from basics.

Python is a general-purpose, versatile and popular programming language. It's great as a first language because it is concise and easy to read, and it is also a good language to have in any programmer's stack as it can be used for everything from web development to software development and scientific applications.

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.



# Python Features

## Easy to Learn

- It is more expressive means that it is more understandable and readable.
- More emphasis on the solution of the problem rather than the syntax

## High-level Language

- In Python, no need to take care about low-level details such as managing the memory used by the program.

## Compiled Interpreted and Platform Independent

- Internally, Python converts the source code into an intermediate form called byte code.
- This byte code is then executed by interpreter on PVM (Python virtual machine).

## Free and Open Source

- Python language is freely available at official web address.
- The source-code is also available. Therefore it is open source.

## Hybrid language

- Supports Procedural as well as Object Oriented Programming.

## Embeddable

- Python can be used within C/C++ program i.e. in other languages.

## Robust

- Exception handling features
- Memory management techniques in built.

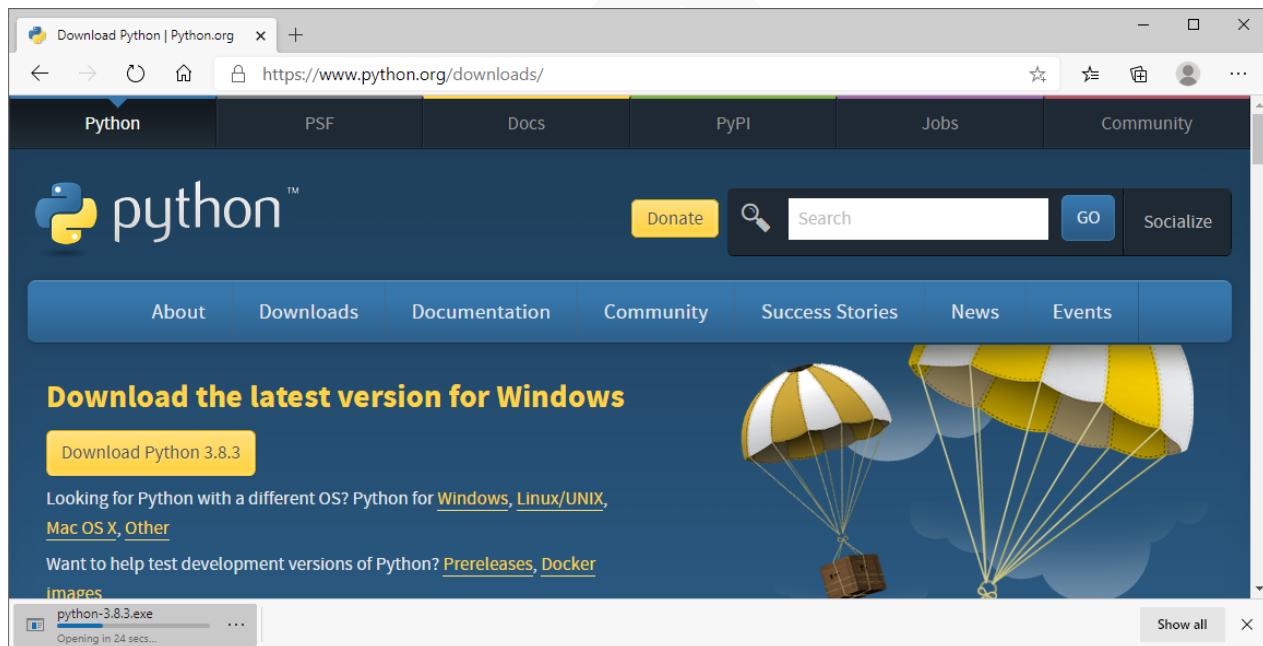
## Rich Library Support

- The Python Standard Library is very vast.
- Library available for NLP, multi-threading, databases, CGI, email, XML, HTML, Image , Audio , Video Processing , cryptography, GUI, Networking and many more.

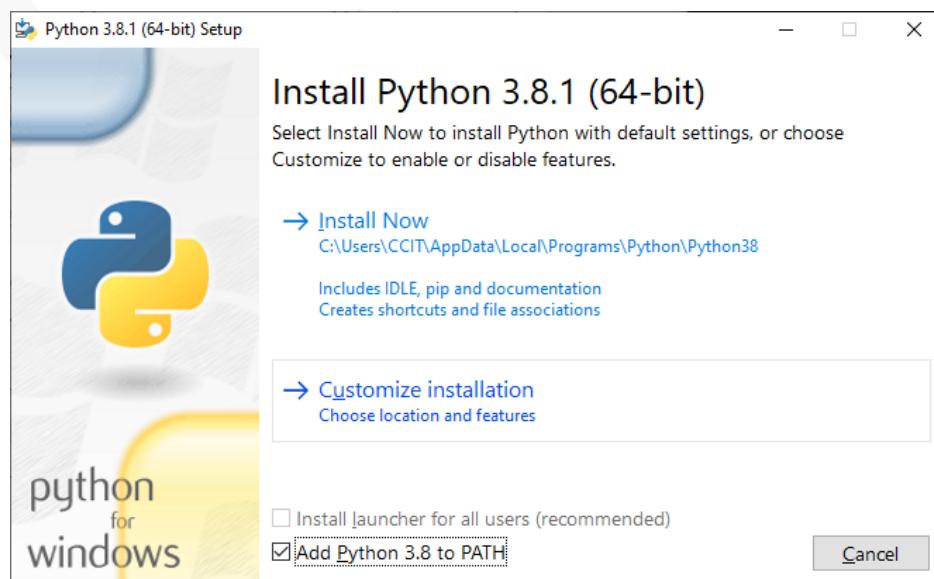
# Python Installation

It is highly unlikely that your Windows system shipped with Python already installed. Windows systems typically do not. Fortunately, installing does not involve much more than downloading the Python installer from the python.org website and running it. Python installers are available for download - two each for the 32-bit and 64-bit versions of the interpreter. The web installer is a small initial download, and it will automatically download the required components as necessary. The offline installer includes the components necessary for a default installation and only requires an internet connection for optional features.

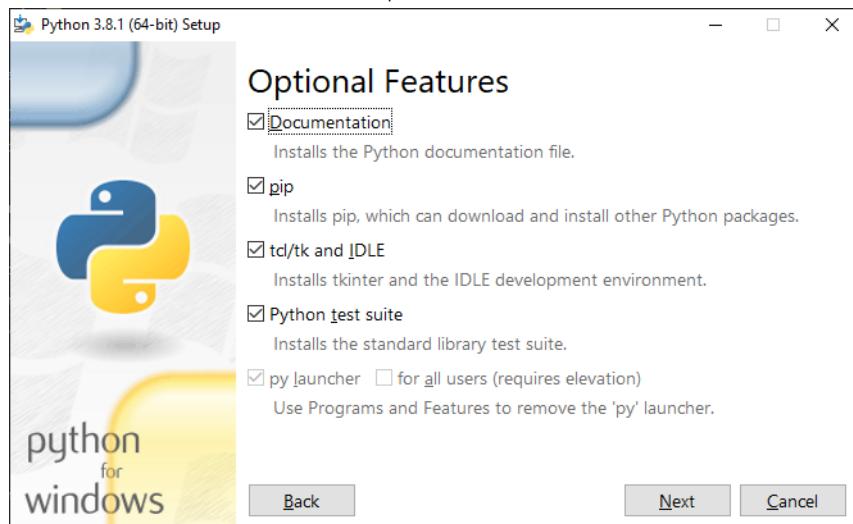
1. Open a Web browser and go to <http://www.python.org/download/>
2. Double click on downloaded installer



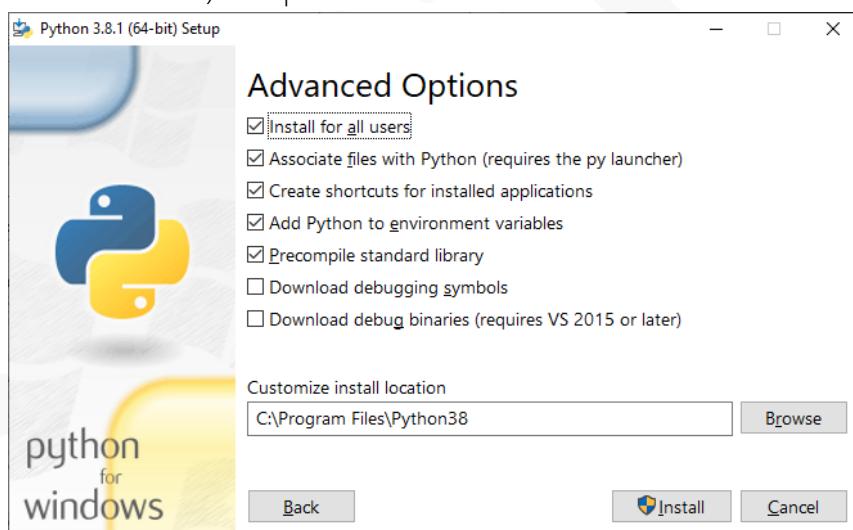
3. Check on (Add python to path). Click on customize installation.



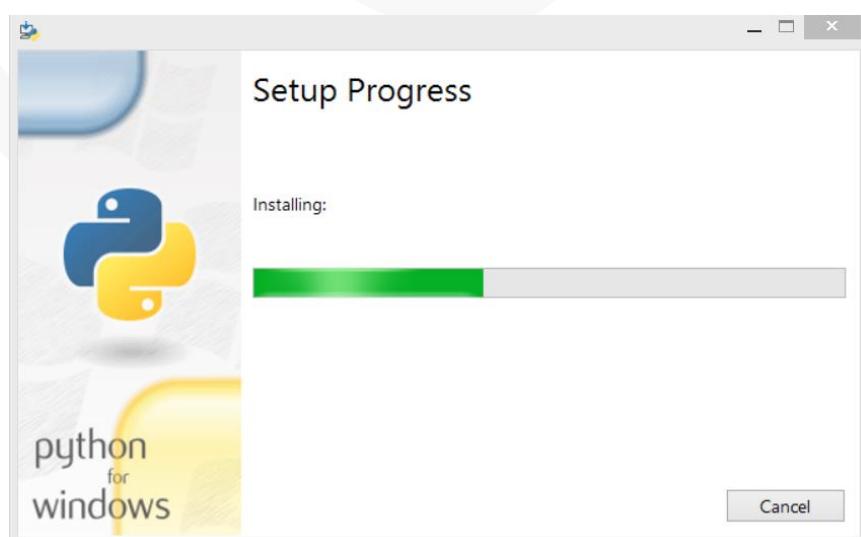
4. Check on all checkboxes shown below and press next button.



5. Check on ( install for all users ) and press install button.



6. It well install python in your system.



# Python IDEs and Code Editors

A code editor is a tool that is used to write and edit code. They are usually lightweight and can be great for learning. However, once your program gets larger, you need to test and debug your code, that's where IDEs come in.

An IDE (Integrated Development Environment) understand your code much better than a text editor. It usually provides features such as build automation, code linting, testing and debugging. This can significantly speed up your work.

1. IDLE



2. PyCharm



3. Visual Studio Code



Visual Studio Code

## IDLE

When you install Python, IDLE is also installed by default. This makes it easy to get started in Python. Its major features include the Python shell window(interactive interpreter), auto-completion, syntax highlighting, smart indentation, and a basic integrated debugger. IDLE is a decent IDE for learning as it's lightweight and simple to use. However, it's not for optimum for larger projects.

 A screenshot of the IDLE Python IDE interface. The title bar says "\*untitled\*". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The main area shows Python code for a bank account system. The code defines two classes: 'Account' and 'CAccount'. The 'Account' class has methods for opening an account, depositing, withdrawing, and showing the balance. The 'CAccount' class inherits from 'Account' and overrides the deposit method to add a service charge. Syntax highlighting is used throughout the code. In the bottom right corner, there is a status bar with "Ln: 20 Col: 0".
 

```

class Account:
    def open(self,an,b1):
        self.accno=an
        self.balance=b1
    def deposit(self,amt):
        self.balance=self.balance+amt
    def withdraw(self,amt):
        self.balance=self.balance-amt
    def showBalance(self):
        print("AccNo is ",self.accno)
        print("Balance is ",self.balance)

class CAccount(Account):
    def deposit(self,amt):
        self.balance = self.balance + amt
        self.balance=self.balance -1
    def withdraw(self,amt):
        self.balance = self.balance - amt
  
```

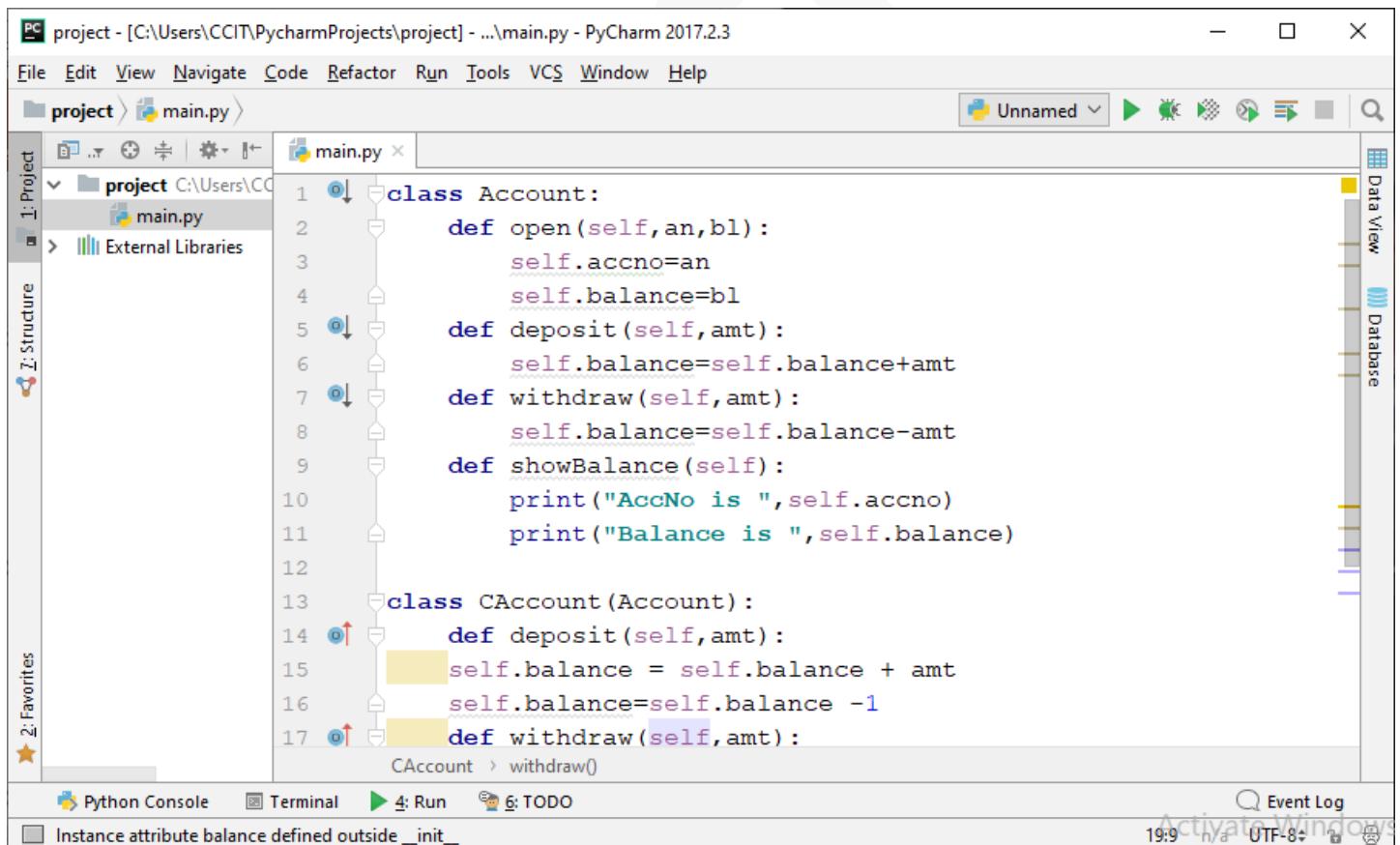
# PyCharm

PyCharm is an IDE for professional developers. It is created by JetBrains, a company known for creating great software development tools.

There are two versions of PyCharm:

- Community - free open-source version, lightweight, good for Python and scientific development
- Professional - paid version, full-featured IDE with support for Web development as well

PyCharm provides all major features that a good IDE should provide: code completion, code inspections, error-highlighting and fixes, debugging, version control system and code refactoring. All these features come out of the box.



The screenshot shows the PyCharm IDE interface. The title bar reads "PC project - [C:\Users\CCIT\PycharmProjects\project] - ...\\main.py - PyCharm 2017.2.3". The menu bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, and Help. The toolbar has icons for Unnamed, Run, Stop, Refresh, and others. The left sidebar has "Project" and "Structure" tabs, showing a project structure with "project" folder containing "main.py" and "External Libraries". The main code editor window displays the following Python code:

```

1  class Account:
2      def open(self,an,bl):
3          self.accno=an
4          self.balance=bl
5      def deposit(self,amt):
6          self.balance=self.balance+amt
7      def withdraw(self,amt):
8          self.balance=self.balance-amt
9      def showBalance(self):
10         print("AccNo is ",self.accno)
11         print("Balance is ",self.balance)
12
13  class CAccount(Account):
14      def deposit(self,amt):
15          self.balance = self.balance + amt
16      def withdraw(self,amt):
17          self.balance=self.balance -1
    CAccount > withdraw()
  
```

The code editor has syntax highlighting and code completion. The status bar at the bottom shows "Python Console", "Terminal", "Run", "TODO", "Event Log", "Activate Windows", "19.9", "UTF-8+", and "15".

# Visual Studio Code

Visual Studio Code (VS Code) is a free and open-source IDE created by Microsoft that can be used for Python development.

You can add extensions to create a Python development environment as per your need in VS code. It provides features such as intelligent code completion, linting for potential errors, debugging, unit testing and so on.

VS Code is lightweight and packed with powerful features. This is the reason why it becoming popular among Python developers.

The screenshot shows a Visual Studio Code interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** demo.py - Python - Visual Studio Code.
- Sidebar:** EXPLORER, OPEN EDITORS (demo.py), PYTHON, OUTLINE, TIMELINE, MAVEN PROJECTS.
- Code Editor:** The file demo.py contains the following Python code:

```
class Account:  
    def open(self,an,bl):  
        self.accno=an  
        self.balance=bl  
    def deposit(self,amt):  
        self.balance=self.balance+amt  
    def withdraw(self,amt):  
        self.balance=self.balance-amt  
    def showBalance(self):  
        print("AccNo is ",self.accno)  
        print("Balance is ",self.balance)  
  
class CAccount(Account):  
    def deposit(self,amt):  
        self.balance=self.balance+amt  
        self.balance=self.balance -1  
    def withdraw(self,amt):
```
- Terminal:** The terminal shows the output of running the script:

```
C:\Users\CCIT\Desktop\Python>C:/Python/python.exe c:/Use  
AccNo is 4117  
Balance is 28000  
AccNo is 3012  
Balance is 9997
```
- Status Bar:** Activate Window, Ln 5, Col 27, Spaces: 4, UTF-8, CRLF, Python, SP Live.

# General Format of python Program

- **Import statement**
- **Global declaration**
  - variables**
  - functions**
  - classes**
- **Statements**

## 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(*objects,sep=' ',end='\n',file=sys.stdout,flush=False)
```

Arguments	Discription
<b>objects</b>	Object to the printed. * indicates that there may be more than one object
<b>separator</b>	Optional. objects are separated by separator. Default value: ''
<b>end</b>	Optional. end is printed at last Default is '\n'.
<b>file</b>	Optional. Must be an object with write(string) method. If omitted it, sys.stdout will be used which prints objects on the screen.
<b>flush</b>	Optional. If True, the stream is forcibly flushed. Default value: False

### For Example:

#### To print string

```
print("Welcome to Python")
>>>Welcome to Python
```

### To print value store in variable and String

```
A="Python"  
print("Welcome to",A)  
>>>Welcome to Python
```

### To print value store in variable

```
A="Ram"  
B="Seeta"  
print(A , B)  
>>>Ram Seeta
```

### To print formatted String

```
A="Python"  
print(f"Welcome to {A}")  
>>>Welcome to Python
```

### To print formatted String

```
A="Welcome"  
B="Python"  
print("{0} to {1}".format(A,B))  
>>>Welcome to Python
```

### To print formatted String

```
A="Welcome"  
B="Python"  
print("%s to %s"%(A,B))  
>>>Welcome to Python
```

### To print separator String

```
A="Ram"  
B="Seeta"  
print(B,A,sep="->")  
>>>Seeta->Ram
```

# Variables & Data Types

# Variables

---

A variable is a location used to store data in the memory. It is helpful to think of variables as a container that holds data which can be changed later throughout programming. In Python, we don't need to specify the type of variable because Python is a dynamically-typed. It automatically decides the datatype depending on type of value.

## Syntax:

```
Variable_Name = Value ;
```

## For Example:

```
A=27  
B="CCIT"  
C=True
```

# Data Types

---

Variables can store data of different types, and different types can do different things. Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. There are various data types in Python. Some of the important types are listed below.

<b>Text Type</b>	String
<b>Numbers</b>	Integer , Float, complex
<b>Sequence Types</b>	List, Tuple ,Sets
<b>Mapping Type</b>	Dictionary
<b>Boolean Type</b>	Bool
<b>Binary Types</b>	Bytes, Bytearray

# Identifiers

---

Python Identifier is the name we give to identify a variable, function, class, module or other object. That means whenever we want to give an entity a name, that's called identifier.

## Rules:

- Identifier must start with a letter or the underscore character
- Next can be alphabets digits or underscores.
- Variable names are case-sensitive.
- No max limit.

# Keywords

---

Python keywords are the words that are reserved. That means you can't use them as name of any entities like variables, classes and functions. So you might be thinking what these keywords are for. They are for defining the syntax and structures of Python language.

You should know there are 33 keywords in Python programming language as of writing this tutorial. Although the number can vary in course of time. Also keywords in Python is case sensitive. So they are to be written as it is. Here is a list of all keywords in python programming.

<b>false</b>	<b>none</b>	<b>in</b>	<b>while</b>	<b>assert</b>
<b>def</b>	<b>import</b>	<b>try</b>	<b>as</b>	<b>finally</b>
<b>if</b>	<b>return</b>	<b>and</b>	<b>except</b>	<b>nonlocal</b>
<b>raise</b>	<b>true</b>	<b>else</b>	<b>lambda</b>	<b>yield</b>
<b>del</b>	<b>elif</b>	<b>is</b>	<b>with</b>	<b>pass</b>
<b>break</b>	<b>not</b>	<b>class</b>	<b>from</b>	
<b>for</b>	<b>or</b>	<b>continue</b>	<b>global</b>	

# Numbers Datatype

Data types that store numeric values are called Number. If you change the value of a number data type, this results in a newly allocated object. So you can call numbers immutable. We can simply create a number object by assigning some value to a variable.

Python supports three different Number data types:

## Int

Integer, is a whole number, positive or negative, without decimals, of unlimited length (from python 3.0 ).

For ex:

1234, -24, 0, 9999999999999999

0o177, 0x9ff, 0b101010

Prefix	Description	Base	Example
0b or 0B	Binary Number	2	0b0101 , 0b0011
0o or 0O	Octal Number	7	0o134, 0o017
Nothing	Decimal Number	10	12, 34 , -5
0x or 0X	Hexadecimal Number	16	0x9ff , 0x1AC

## Float

Float, or "floating point number" is a number, positive or negative, containing decimal point.

For ex:

1.23, 1.

3.14e-10, 4E210, 4.0e+210

## Complex

Complex numbers are written with a "j" as the imaginary part.

For ex:

3+4j, 3.0+4.0j, 3j

# String Datatype

The string is a sequence of characters. Python supports Unicode characters. Generally, strings are represented by either single or double quotes. Python treats single quotes the same as double quotes.

String Quotes	Example
Single quotes	'Welcome to CCIT'
Double quotes	"Welcome to CCIT"
Triple quotes	<pre>'''Welcome to CCIT Amravati'''</pre> <pre>"""Welcome to CCIT Amravati"""</pre>

Single and Double quotes

Single and double quotes are used for single line string.

Triple quotes

Triple quotes are used for multiline string.

# Boolean Datatype

Booleans represent one of two values: True or False. In programming you often need to know if an expression is True or False. You can evaluate any expression in Python, and get one of two answers, True or False.

Boolean Datatype	True and False
------------------	----------------

# Operator

---

---

# Operators

An operator accepts one or more inputs in the form of variables or expressions, performs a task (such as comparison or addition), and then provides an output consistent with that task. Operators are classified partially by their effect and partially by the number of elements they require. For example, a unary operator works with a single variable or expression; a binary operator requires two.

## Unary Operator

Unary operators require a single variable or expression as input. You often use these operators as part of a decision-making process. For example, you might want to find something that isn't like something else.

Operator	Description	Example
<code>~</code>	Inverts the bits in a number so that all the 0 bits become 1 bits and vice versa.	<code>~4 =&gt; -5</code>
<code>-</code>	Negates the original value so that positive becomes negative and vice versa.	<code>-(-4) =&gt; 4</code> <code>-4 =&gt; -4</code>
<code>+</code>	Is provided purely for the sake of completeness. This operator returns the same value that you provide as input.	<code>+4 =&gt; 4</code>

## Arithmetic Operator

Computers are known for their capability to perform complex math. However, the complex tasks that computers perform are often based on much simpler math tasks, such as addition.

Operator	Description	Example
<code>+</code>	Adds two values together	<code>5 + 2 =&gt; 7</code>
<code>-</code>	Subtracts the right operand from the left operand	<code>5 - 2 =&gt; 3</code>
<code>*</code>	Multiplies the right operand by the left operand	<code>5 * 2 =&gt; 10</code>
<code>/</code>	Divides the left operand by the right operand	<code>5 / 2 =&gt; 2.5</code>
<code>%</code>	Divides the left operand by the right operand and returns the remainder	<code>5 % 2 =&gt; 1</code>
<code>**</code>	Calculates the exponential value of the right operand by the left operand	<code>5 ** 2 =&gt; 25</code>
<code>//</code>	Performs integer division, in which the left operand is divided by the right operand and only the whole number is returned (also called floor division)	<code>5 // 2 =&gt; 2</code>

## Relational Operator

The relational operators compare one value to another and tell you when the relationship you've provided is true.

Operator	Description	Example
<code>==</code>	Determines whether two values are equal.	<code>1 == 2</code> is <code>False</code>
<code>!=</code>	Determines whether two values are not equal.	<code>1 != 2</code> is <code>True</code>
<code>&gt;</code>	Verifies that the left operand value is greater than the right operand value.	<code>1 &gt; 2</code> is <code>False</code>
<code>&lt;</code>	Verifies that the left operand value is less than the right operand value.	<code>1 &lt; 2</code> is <code>True</code>
<code>&gt;=</code>	Verifies that the left operand value is greater than or equal to the right operand value	<code>1 &gt;= 2</code> is <code>False</code>
<code>&lt;=</code>	Verifies that the left operand value is less than or equal to the right operand value.	<code>1 &lt;= 2</code> is <code>True</code>

## Logical Operator

The logical operators combine the true or false value of variables or expressions so that you can determine their resultant truth value.

Operator	Description	Example
<code>and</code>	Determines whether both operands are true.	True and True is <code>True</code> True and False is <code>False</code> False and True is <code>False</code> False and False is <code>False</code>
<code>or</code>	Determines when one of two operands is true.	True or True is <code>True</code> True or False is <code>True</code> False or True is <code>True</code> False or False is <code>False</code>
<code>not</code>	Negates the truth value of a single operand. A true value becomes false and a false value becomes true.	not True is <code>False</code> not False is <code>True</code>

## Assignment Operator

The assignment operators place data within a variable. Python offers a number of other interesting assignment operators that you can use. These other assignment operators can perform mathematical tasks during the assignment process, which makes it possible to combine assignment with a math operation.

Operator	Description	Example
=	Assigns the value found in the right operand to the left operand.	A = 2 results in A containing 2
+=	Adds the value found in the right operand to the value found in the left operand and places the result in the left operand.	a = 5 a += 2 results in a containing 7
-=	Subtracts the value found in the right operand from the value found in the left operand and places the result in the left operand.	a = 5 a -= 2 results in a containing 3
*=	Multiplies the value found in the right operand by the value found in the left operand and places the result in the left operand.	a = 5 a *= 2 results in a containing 10
/=	Divides the value found in the left operand by the value found in the right operand and places the result in the left operand.	a = 5 a /= 2 results in a containing 2.5
%=	Divides the value found in the left operand by the value found in the right operand and places the remainder in the left operand.	a = 5 a %= 2 results in a containing 2
**=	Determines the exponential value found in the left operand when raised to the power of the value found in the right operand and places the result in the left operand.	a = 5 a **= 2 results in a containing 25
//=	Divides the value found in the left operand by the value found in the right operand and places the integer (whole number) result in the left operand.	a = 5 a //= 2 results in a containing 2

## Bitwise Operator

A bitwise operator would interact with each bit within the number in a specific way. When working with a logical bitwise operator, a value of 0 counts as false and a value of 1 counts as true.

Operator	Description	Example
<b>&amp; (and)</b>	Determines whether both individual bits within two operators are true and sets the resulting bit to true when they are.	0b1100 & 0b0110 = <b>0b0100</b>
<b>  (or)</b>	Determines whether either of the individual bits within two operators is true and sets the resulting bit to true when one of them is.	0b1100   0b0110 = <b>0b1110</b>
<b>^ (Exclusive or)</b>	Determines whether just one of the individual bits within two operators is true and sets the resulting bit to true when one is. When both bits are true or both bits are false, the result is false.	0b1100 ^ 0b0110 = <b>0b1010</b>
<b>~ (One complement)</b>	Calculates the one's complement value of a number.	$\sim 0b1100 = \textcolor{green}{\sim 0b1100}$ $\sim 0b0110 = \textcolor{green}{\sim 0b0111}$
<b>&lt;&lt; (Left shift)</b>	Shifts the bits in the left operand left by the value of the right operand. All new bits are set to 0 and all bits that flow off the end are lost.	0b00110011 << 2 = <b>0b11001100</b>
<b>&gt;&gt; (Right shift)</b>	Shifts the bits in the left operand right by the value of the right operand. All new bits are set to 0 and all bits that flow off the end are lost.	0b00110011 >> 2 = <b>0b00001100</b>

## Membership Operator

The membership operators detect the appearance of a value within a list or sequence and then output the truth value of that appearance.

Operator	Description	Example
<b>in</b>	Determines whether the value in the left operand appears in the sequence found in the right operand.	"CCIT" in "Welcome to CCIT" is <b>True</b>
<b>not in</b>	Determines whether the value in the left operand is missing from the sequence found in the right operand.	"CCIT" not in "Welcome to CCIT" is <b>False</b>

## Operator precedence

When you create simple statements that contain just one operator, the order of determining the output of that operator is also simple. However, when you start working with multiple operators, it becomes necessary to determine which operator to evaluate first.

Operator	Description
( )	You use parentheses to group expressions and to override the default precedence so that you can force an operation of lower precedence (such as addition) to take precedence over an operation of higher precedence (such as multiplication).
**	Exponentiation raises the value of the left operand to the power of the right operand.
~ + -	Unary operators interact with a single variable or expression.
* / % //	Multiply, divide, modulo, and floor division.
+ -	Addition and subtraction.
>> <<	Right and left bitwise shift.
&	Bitwise AND.
^	Bitwise exclusive OR and standard OR
<= < > >=	Comparison operators.
== !=	Equality operators.
= %= /= //=- -= += *= **=	Assignment operators.
In not in	Membership operators.
not or and	Logical operators

# Multiple Statements

We could also put multiple statements in a single line using semicolons, as follows

## For Example:

```
a = 1; b = 2; c = 3
```

# Multi-Line Statements

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\).

## For Example:

```
S=1+2+3+\n    4+5+6+\n    7+8+9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

## For Example:

```
a = (1 + 2 + 3 +\n      4 + 5 + 6 +\n      7 + 8 + 9)\ncolors = ['red',\n           'blue',\n           'green']
```

# Multiple Assignments

Python allows us to assign values to multiple variables in a single statement.

## Syntax:

```
var1, var2, var3.. = value1, value2, value3..
```

## For Example:

```
a, b ,c = 4 ,3 ,2
```

# Comments

In Python there are two types of comments- Single line comments and multiple lines comments. Single line commenting is commonly used for a brief and quick comment (or to debug a program, we will see it later). On the other hand we use the multiple lines comments to note down something much more in details or to block out an entire chunk of code.

## Single line comments

Python single line comment starts with hashtag symbol with no white spaces (#) and lasts till the end of the line.

### For Example:

```
# This is a comment  
# Print "Welcome !" to console
```

## Multi-line comments

Python multi-line comment is a piece of text enclosed in a delimiter (""""") on each end of the comment. Quotes can be single or double

### For Example:

```
"""This is a  
multi-line comment.  
We are printing welcome """
```

# Type Casting

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python provides type conversion functions to directly convert one data type to another data type.

## Functions

### **bin()**

Converts an integer to a binary string

### **bool()**

Converts an argument to a Boolean value

### **complex()**

Returns a complex number constructed from arguments

### **float()**

Returns a floating-point object constructed from a number or string

### **hex()**

Converts an integer to a hexadecimal string

### **int()**

Returns an integer object constructed from a number or string

### **oct()**

Converts an integer to an octal string

### **ord()**

Returns integer representation of a character

### **str()**

Returns a string version of an object

# Input Function

Python has an input function which lets you ask a user for some text input. You call this function to tell the program to stop and wait for the user to key in the data. The program will resume once the user presses the ENTER or RETURN key. This function is used to read user input.

Note: `input()` returns the string that is given as user input. It will return entered value as string.

## Syntax:

```
Variable_Name = Input('Message');
```

## For Example:

### To Read User Name as string

```
A=input("Enter Your Name")
>>> Enter Your Name Amit Jain
```

### To Read a integer Value

```
A=int(input("Enter a Number"))
>>> Enter a Number 23
```

### To Read three Value

```
print("Enter 3 Number")
a,b,c = input(),input(),input()

>>> Enter 3 Number
>>> 5
>>> 2
>>> 3
```

To Read three Value

```
a,b,c=input("Enter 3 Number").split(' ')
>>> Enter 3 Number 5 2 3
```

To Read three integer Value

```
a,b,c=map(int,input("Enter 3 Number ").split(','))

>>> Enter 3 Number 5,2,3
```

To Read three integer Value

```
a,b,c=map(int,input("Enter 3 Number ").split(' '))

>>> Enter 3 Number 5 2 3
```

To Read Multiple Value as integer in list

```
a=list(map(int,input("Enter Numbers ").split(' ')))

>>> Enter Numbers 5 12 3 5 68 1
```

To Read Multiple Value as integer in list

```
a=list(map(str,input("Enter Students Name ").split(' ')))

>>> Enter Students Name Amit Sumit Gopal Arjun
```

**Examples :****Program to read 2 numbers and find their sum.**

Program

```
a=input("Enter a no ")
b=input("Enter a no ")
c=int(a) + int(b)
print("Result is ",c)
```

Output

```
Enter a no 2
Enter a no 3
Result is 5
```

**Examples :****Program to read 2 numbers and find their sum.**

Program

```
a=int(input("Enter a no "))
b=int(input("Enter a no "))
c=a+b
print("Result is ",c)
```

Output

```
Enter a no 2
Enter a no 3
Result is 5
```

**Examples :****WAP to read a radius of circle and find its area and circumference.**

Program

```
r=input("Enter Radius ")
a=3.14*r**2
c=2*3.14*r
print("Area is ",a)
print("Circumference is ",c)
```

Output

```
Enter Radius 5
Area is 78.5
Circumference is 31.4
```

## Examples :

### WAP to exchange values of 2 variables

Program

```
a,b=input("Enter 2 Number ").split(' ')
print("A=%s B=%s ",%(a,b))
a,b=b,a
print("A=%s B=%s ",%(a,b))
```

Output

```
Enter 2 Number 5 2
A=5 B=2
A=2 B=5
```

## Examples :

### WAP to convert Fahrenheit to Celsius.

Program

```
a=int(input("Enter Temperature"))
c=5/9*(a-32)
print("Temperature in Celsius",c)
```

Output

```
Enter Temperature 104
Temperature in Celsius 40
```

# Condition &

# looping

# Structure

# if...else Statements

Decision making is required when we want to execute a code only if a certain condition is satisfied. The if...else statement is used in Python for decision making. It is used to conditionally execute statements. The else statement allows you to define an action or set of actions that are executed when the conditional test fails.

## Syntax:

```
if condition:
    statements
    -----
else:
    statements
    -----
```

## Boolean expressions

Condition can be specified by using a Boolean expression i.e. an expression whose result is True or False. Boolean expression can be created by using relational and logical operators. A Boolean expression is just another name for a conditional test. A Boolean value is either True or False, just like the value of a conditional expression after it has been evaluated.

Relational Operators: < , > , <= , >= , == , !=

Logical Operators: and , or , not

**Examples :****WAP to read a number and check if it is an even no or odd no**

Program

```
a=int(input("Enter a Number "))

if a%2==0:

    print("Number is Even")

else:

    print("Number is Odd")
```

Output

```
Enter a Number 24
Number is Even
```

**Examples :****WAP to read 3 angles and check if triangle can be formed or not.**

Program

```
a,b,c=input("Enter 3 Angles").split(' ')
if int(a)+int(b)+int(c)==180:

    print("Triangle can be formed")

else:

    print("Triangle cannot formed")
```

Output

```
Enter 3 Angles 45 90 45
Triangle can be formed
```

# if...elif...else Statements

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition. The if block can have only one else block. But it can have multiple elif blocks.

## Syntax:

```
if condition:  
    statements  
    -----  
    elif condition:  
        statements  
        -----  
        elif condition:  
            statements  
            -----  
            .  
            .  
    else:  
        statements  
        -----
```

**Examples :****WAP to read 3 different numbers and find greatest of them.****Program**

```
a,b,c=input("Enter 3 Number").split(' ')
a,b,c=int(a),int(b),int(c)
if a>b and a>c:
    print(a,"is Greatest")
elif b>c:
    print(b,"is Greatest")
else:
    print(c" is Greatest")
```

**Output**

```
Enter 3 Number 45 90 30
90 is Greatest
```

**Examples :****WAP to read a number and check if it is Positive , Negative or Zero.****Program**

```
a=int(input("Enter a Number "))
if a>0:
    print("Positive")
elif a<0:
    print("Negative")
else:
    print("Zero")
```

**Output**

```
Enter a Number -12
Negative
```

# Ternary Operator

Python ternary operator is also termed as conditional operator. This is because it can evaluate a statement with a condition being true or false used properly, ternary operator can reduce code size and increase readability of the code. There is no special keyword for ternary operator, it's the way of writing if-else statement that creates a ternary statement or conditional expression. If condition is true then statement1 will be evaluated else statement2 will be evaluated.

## Syntax:

```
Variable-name=statement1 if condition else statement2
```

## Examples :

**WAP to read 2 numbers and find greatest of them.**

### Program

```
a=int(input("Enter a Number "))
b=int(input("Enter a Number "))
G=a if a>b else b
print(G,"is greatest")
```

### Output

```
Enter a Number 12
Enter a Number 23
23 is greatest
```

## Examples :

**WAP to read 4 numbers and find greatest of them.**

### Program

```
a,b,c,d=input("Enter 4 Number ").split(' ')
a,b,c,d=int(a),int(b),int(c),int(d)
x=a if a>b else b
y=c if c>d else d
G=x if x>y else y
print(G,"is greatest")
```

### Output

```
Enter 4 Number 2 8 6 4
8 is greatest
```

# while Loop

Python while loop is used to repeatedly execute some statements until the condition is true. The while statement works with a condition rather than a sequence. The condition states that the while statement should perform a task until the condition is no longer true. First condition is checked If condition is true, then statement within while loop are executed and again condition is checked The above process is repeated while the condition is true. Program control is transferred to next statements only when condition becomes false.

## Syntax:

```
while condition:
    statement
    -----
    -----
```

## Examples :

### WAP to print all number from 1 to 10

#### Program

```
i=1
while i<=10:
    print(i)
    i=i+1
```

#### Output

```
1
2
3
4
5
6
7
8
9
10
```

## Examples :

### WAP to print all number from 1 to 10

#### Program

```
i=1
while i<=10:
    print(i,end=" ")
    i=i+1
```

#### Output

```
1 2 3 4 5 6 7 8 9 10
```

**Examples :****WAP to print all odd number from 1 to 50**

Program

```
i=1
while i<=10:
    print(i,end=" ")
    i=i+2
```

Output

```
1 3 5 7 9 11 13 15 17
19 21 23 25 27 29 31 33
35 37 39 41 43 45 47 49
```

**Examples :****WAP to read a number print all even numbers from 1 to given number**

Program

```
n=int(input('Enter a number'))
i=1
while i<=n:
    if i%2==0:
        print(i,end=" ")
    i=i+1
```

Output

```
Enter a number 26
2 4 6 8 10 12 14 16 18
20 22 24 26
```

**Examples :****WAP to read a number and find sum of digits**

Program

```
n=int(input("Enter a Number"))
s=0
while n>0:
    l=n%10
    n=n//10
    s=s+l
print("Sum of digits is ",s)
```

Output

```
Enter a Number 1234
Sum of digits is 10
```

# while...else Loop

Python while loop is used to repeatedly execute some statements until the condition is true. The while statement works with a condition rather than a sequence. Else block will execute when condition becomes false. If loop is terminated without condition becoming false then else will not be executed.

## Syntax:

```
while condition:
```

```
    statement
```

```
    -----
```

```
else:
```

```
    statement
```

```
    -----
```

## Examples :

### WAP to read a number and check number is prime no. or not

#### Program

```
n=int(input("Enter a no."))
i=2
while i<n:
    if n%i==0:
        print("No. is Not Prime ")
        break
    i=i+1
else:
    print("No. is Prime")
```

#### Output

```
Enter a no. 5
No. is Prime
```

# for Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Loop continues until we reach the last item in the collection. The body of for loop is separated from the rest of the code using indentation. It is used to process all elements of a collection. Statements within for loop are repeatedly executed for each value of collection

## Syntax:

```
for item in collection:
    statement
    -----
    -----
```

## Examples :

### WAP to print all numbers in a list collection

#### Program

```
arr=[10,50,60,20,30]
s=0
for n in arr:
    print(n)
```

#### Output

```
10
50
60
20
30
```

## Examples :

### WAP to find sum of all numbers in a list collection

#### Program

```
arr=[10,50,60,20,30]
s=0
for n in arr:
    s=s+n

print("Sum is ",s)
```

#### Output

```
Sum is 170
```

**Examples :****WAP to find sum of all even numbers in a list collection**

Program

```
arr=[10,15,16,22,32,11,4,1]
s=0
for n in arr:
    if n%2==0:
        s=s+n
print("Sum of even nos is ",s)
```

Output

Sum of even is 82

**Examples :****WAP to count all even numbers in a list collection**

Program

```
arr=[10,15,16,22,32,11,4,1]
c=0
for n in arr:
    if n%2==0:
        c=c+1
print("Total number of even are ",c)
```

Output

Total number of even  
are 5

**Examples :****WAP to count all odd numbers in a list collection**

Program

```
arr=[10,15,16,22,32,11,4,1]
c=0
for n in arr:
    if n%2!=0:
        c=c+1
print("Total number of odd are ",c)
```

Output

Total number of odd are  
3

# range function

It returns a Collection containing sequence of number. These values can be used for looping. To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as range(start, stop, step size). step size defaults to 1 if not provided.

## Syntax:

```
range(start ,stop ,step)  

range(start ,stop)  

range(stop)
```

## Examples :

### WAP to print all numbers from 1 to 10

#### Program

```
for n in range(1,11):
    print(n ,end=" ")
```

#### Output

```
1 2 3 4 5 6 7 8 9 10
```

## Examples :

### WAP to print all even numbers from 1 to 100

#### Program

```
for n in range(2,101,2):
    print(n ,end=" ")
```

#### Output

```
2 4 6 8 10 12 14 16 18
20 22 24 26 28 30 32 34
36 38 40 42 44 46 48 50
52 54 56 58 60 62 64 66
68 70 72 74 76 78 80 82
84 86 88 90 92 94 96 98
100
```

**Examples :****WAP to read a number and print all even numbers from 1 to given number**

Program

```
n=int(input("Enter A Number"))

for n in range(2,n+1):
    print(n ,end=" ")
```

Output

Enter a Number 28

2 4 6 8 10 12 14 16 18  
20 22 24 26 28**Examples :****WAP to read a number and print all numbers from given number to 1 which are divisible 2 or 3**

Program

```
n=int(input("Enter A Number"))

for n in range(n,0,-1):
    print(n ,end=" ")
```

Output

Enter a Number 16

16 15 14 12 10 9 8 6 4 3  
2**Examples :****WAP to read a number and find factorial of given number**

Program

```
n=int(input("Enter A Number"))

f=1

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

    f=f*i

print(f"Factorial of {n} is {f}")
```

Output

Enter a Number 5

Factorial of 5 is 120

**Examples :****WAP to print all chars from string**

Program

```
w=input("Enter A String")
for i in w:
    print(i)
```

Output

```
Enter a String Amravati
A
m
r
a
v
a
t
i
```

**Examples :****WAP to count total number of words in string**

Program

```
w=input("Enter a String")
c=1
for i in w:
    if i==" ":
        c=c+1
print("Total number of word are ",c)
```

Output

```
Enter a String
Welcome to CCIT Amravati

Total number of word are
4
```

**Examples :****WAP to count total number of vowels in string**

Program

```
s=input("Enter a String")
c=0
for i in s:
    if ( i=='A' or i=='a' or
        i=='E' or i=='e' or
        i=='I' or i=='i' or
        i=='O' or i=='o' or
        i=='U' or i=='u'):
        c=c+1
print("Total number of vowels are ",c)
```

Output

```
Enter a String
Welcome to CCIT

Total number of vowels
are 5
```

# for...else Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Loop continues until we reach the last item in the collection. For loop is used to process all elements of a collection. Statements within for loop are repeatedly executed for each value of collection else block will execute when all items has been processed. if loop is terminated in between then else will not be executed.

## Syntax:

```
for item in collection:
    statement
    -----
    else:
        statement
    -----
```

## Examples :

### WAP to read a number and check number is prime no. or not

#### Program

```
lst=[2,4,5,12,13,8]
n=input("Enter number to search")
n=int(n)
for item in lst:
    if n==item:
        print("Found ..")
        break
else:
    print("Not Found")
```

#### Output

```
Enter number to search
12

Found..
```

# Break Statement

Break statement is used to exit from the iterative statements (loops) such as for, while. Break statement terminates the execution of loop immediately, and then program execution will jump to the next statements. The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

## Syntax:

```
break
```

## Examples :

### WAP to read a number and check number is prime no. or not

#### Program

```
n=int(input("Enter a no."))

i=2

while i<n:

    if n%i==0:

        print("No. is Not Prime ")

        break

    i=i+1

else:

    print("No. is Prime")
```

#### Output

```
Enter a no. 5

No. is Prime
```

# Continue Statement

Continue statement is used to continue the loop execution. It transfer program control to start of loop to retest condition skipping any statements after continue. The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

## Syntax:

```
continue
```

## Examples :

**WAP to read a number and print all odd number from 1 to given number.**

### Program

```
n= int(input("Enter a Number"))

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

    if i%2==0:

        continue

    print(i)
```

### Output

```
Enter a Number

24

1 3 5 7 9 11 13 15 17 19
21 23
```

# pass Statement

pass statement is used when programmer don't want to execute statement pass statement is null operation. Nothing will happen when pass statement has been executed. We will use the pass statement when we don't want to execute the code, but want the syntactical expressions. In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).

## Syntax:

```
pass
```

## Examples :

### Program

```
n= int(input("Enter a Number"))

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

    pass
```

### Output

```
Enter a Number
24
```

# Functions

---

---

# Function

A Function is a block of code designed to perform some specific task. Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code. In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

## Syntax:

```
def <function-name>():
    Statement
    -----
```

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

## Syntax:

```
<function-name>(argument, argument...)
```

## Examples :

### Program

```
def star():
    for i in range(1,21):
        print("*",end="")
print("CCIT")
star()
print("\n Amravati")
star()
```

### Output

```
CCIT
*****
Amravati
*****
```

# Function with Arguments

A Function is a block of code designed to perform some specific task. While calling function we can pass some data (actual parameters). This data is passed to function as arguments (formal parameters). According to arg value received function can perform different task.

## Syntax:

```
def <function-name>(parameters1,parameters2...):  
    Statement  
    -----
```

## Examples :

### Program

```
def star(n):  
    for i in range(1,n+1):  
        print("*",end="")  
  
print("CCIT")  
star(10)  
print("\n Amravati")  
star(20)
```

### Output

```
CCIT  
*****  
  
Amravati  
*****
```

**Examples :**

**Design a function interest which will calculate and print simple interest from 3 arguments p,r,n.**

Program

```
def interest(p,r,t):
    si=(p*r*t)/100
    print("simple interest is ",si)
interest(15200,12.45,5)
```

Output

Simple interest is 9462

**Examples :**

**Design a function volume which will calculate and print volume from 3 arguments l,b,h.**

Program

```
def volume(l,b,h):
    v=l*b*h
    print("Volume is ",v)
volume(15.2,5.5,5)
```

Output

Volume is 418

**Examples :**

**Design a function digitsum which will take one value as argument and find sum of its digits**

Program

```
def digitsum(n):
    s=0
    while n>0:
        l=n%10
        n=n//10
        s=s+l
    print("Sum of digits ",s)
digitsum(1234)
```

Output

Sum of digits 10

# Function with Default Arguments

A Function is a block of code designed to perform some specific task. While defining a function we can assign some default values for its arguments. Only arguments from right side can be assigned default values. In case value is not provided in the function call default value is used.

## Syntax:

```
def <function-name>(parameters1,parameters2...,parameter=value):  
    Statement  
    -----
```

## Examples :

**Design a function interest which will calculate and print simple interest from 3 arguments p,r,n. Use default value t=1**

### Program

```
def interest(p,r,t=1):  
    si=(p*r*t)/100  
    print("simple interest is ",si)  
  
interest(15200,12.45,5)  
interest(2500,15.24)
```

### Output

```
Simple interest is 9462  
Simple interest is 381
```

# Function returning value

A Function is a block of code designed to perform some specific task. If we want to use our function in expression then our function must return value. A function can return value by using a return statement. The return statement returns value at point from where function is call. In case no expression is given after return it will return None. The return statement is used to exit a function and go back to the place from where it was called.

## Syntax:

```
def <function-name>(parameters1,parameters2...):
    Statement
    -----
    return [value]
```

## Examples :

**Design a function interest which will calculate and print simple interest from 3 arguments p,r,n.**

### Program

```
def interest(p,r,t):
    si=(p*r*t)/100
    return si

i=interest(12050,12.45,3)
print("Simple interest is ",i)
```

### Output

Simple interest is 4500.675

**Examples :**

**Design a function fact which will calculate and return factorial of a no. which is pass as argument.**

**Program**

```
def fact(n):
    f=1
    for i in range(1,n+1):
        f*=i
    return f

f=fact(5)
print("Factorial is",f)
```

**Output**

Factorial is 120

**Examples :**

**Design two functions max and min which will calculate and return greatest and smallest of a no. which is pass as argument.**

**Program**

```
def max(a,b):
    if a>b:
        return a
    else:
        return b

def min(a,b):
    if a<b:
        return a
    else:
        return b

print("Greatest is",max(12,32))
print("Smallest is",min(12,32))
```

**Output**

Greatest is 32  
Smallest is 12

# Function Arbitrary Arguments

A Function is a block of code designed to perform some specific task. It is used to pass variable number of arguments to a function. Only one argument of this type is allowed. It can only be last argument. All arguments values are received in a collection of type list. Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with arbitrary number of arguments. In the function definition we use an asterisk (\*) before the parameter name to denote this kind of argument.

## Syntax:

```
def <function-name>(parameters1,parameters2,...,*args):
    Statement
    -----
    return [value]
```

## Examples :

**Design a function sum to find sum of all arguments .**

### Program

```
def Sum(*lst):
    s=0
    for n in lst:
        s+=n
    print("Sum is",s)
```

Sum(12,23)

Sum(32,21,11,22)

### Output

Sum is 35

Sum is 86

# Function Keyword Arguments

A Function is a block of code designed to perform some specific task. The special syntax `**kwargs` in function definitions is used to pass a keyworded, variable-length argument list. i.e to pass variable number of named parameters. Only one argument of this type is allowed. It can only be last argument. `kwargs` is a dictionary in which all name-value pair will be received.

## Syntax:

```
def <function-name>(parameters1,...,**kwargs):
    Statement
    -----
    return [value]
```

## Examples :

**Design a function result to display result of student from given info.**

### Program

```
def result(rollno,name,**marks):
    print("Roll No. :",rollno)
    print("Name :",name)
    t=0
    for s,m in marks.items():
        print(s,":",m)
        t+=m
    p=t/len(marks)
    print("Total Marks :",t)
    print("Percentage :",p)

result(1001,"Amit Jain",
      English=67,
      Math=75,
      Scince=92)
```

### Output

Roll No.	:	1001
Name	:	Amit Jain
English	:	67
Math	:	75
Scince	:	92
Total Marks:		234
Percentage :		78.0

# Recursive Function

A Function is a block of code designed to perform some specific task. Recursion is the process of defining something in terms of itself. A function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions. Recursive functions make the code look clean and elegant. A complex task can be broken down into simpler sub-problems using recursion. Sequence generation is easier with recursion than using some nested iteration.

## Syntax:

```
def <function-name>(parameters1,...):
    Statement
    -----
    <Function-name>()
    -----
    return [value]
```

## Examples :

**Design a function fact which will calculate and return factorial of a no. which is pass as argument.**

### Program

```
def fact(n):
    if n==1:
        return 1
    else:
        return n*fact(n-1)

print("Factorial is ",fact(5))
```

### Output

Factorial is 120

# Function Aliases / References

A Function is a block of code designed to perform some specific task. If required a new name/Reference can be assigned to function just by storing it into another variable.

## Syntax:

```
def <function-name>(parameters1,...):
    Statement
    -----
    -----
    return [value]

<aliases-name>=<function-name>
```

## Examples :

**Design a function fact which will calculate and return factorial of a no. which is pass as argument.**

### Program

```
def fact(n):
    f=1
    for i in range(1,n+1):
        f=f*i
    return f

factorial=fact

print("Factorial is",fact(5))
print("Factorial is",factorial(6))
```

### Output

```
Factorial is 120
Factorial is 720
```

# Nested Functions

A Function is a block of code designed to perform some specific task. If required you can define a function inside another function in Python.

## Syntax:

```
def <function-name>(parameters):
    Statement
    -----
    def <function-name>(parameters):
        Statement
        -----
        -----
    return [value]
```

## Examples :

**Design a function area which will calculate and return area of circle. When radius is pass as argument.**

### Program

```
def area(n):
    def sqr(n):
        return n*n
    a=3.14*sqr(n)
    return a

r=int(input("Enter Radius "))
print("Area is",area(r))
```

### Output

```
Enter Radius 13
Area is 530.66
```

# Functions as Arguments

A Function is a block of code designed to perform some specific task. If required a Function can be passed as argument to other function. When we pass a function as argument actually the reference of function is passed as argument.

## Syntax:

```
def <function-name>(func):
    Statement
    -----
    func()
    -----
    return [value]
```

## Examples :

### Program

```
def sqr(n):
    return n*n

def cube(n):
    return n*n*n

def display(fn):
    for i in range(1,11):
        print(fn(i))

display(sqr)
display(cube)
```

### Output

```
1
4
9
16
25
36
49
64
81
100
1
8
27
64
125
216
343
512
729
1000
```

# Functions Returning Functions

A Function is a block of code designed to perform some specific task. If required A function can return a function.

## Syntax:

```
def <function-name>(parameters):
    Statement
    -----
    def <function-name>(parameters):
        Statement
        -----
        -----
    return <function-name>
```

## Examples :

### Program

```
def display(m):
    def sqr(n):
        return n*n
    def cube(n):
        return n*n*n
    if m==1:
        return sqr
    if m==2:
        return cube

print("Square ",display(1)(12))
print("Cube ",display(2)(12))
```

### Output

```
Square 144
Cube 1728
```

# Function Decorator

A decorator is a design pattern in Python that allows a user to add new functionality to an existing function without modifying its structure. A decorator function is a function that takes a function as its argument and returns a function with additional features. Due to this when this function is called, internally the decorator function will be automatically called and the function is passed as argument.

## Syntax:

```
def <function-name>(function):
    def <function-name>(parameters):
        Statement
        -----
        Statement
        -----
    return <function-name>
```

## Examples :

### Program

```
def star(func):

    def inner(n):
        func(n)
        for i in range(1,20):
            print("*",end=' ')
        print()
    return inner

@star
def message(m):
    print(m)

message("Welcome to CCIT")
```

### Output

```
Welcome to CCIT
*****
Welcome to Python
*****
```

## Examples :

### Program

```
def nonnegative(fn):
    def inner(n):
        if n>0:
            return fn(n)
        else:
            return "Value must be >0"
    return inner

@nonnegative
def isEven(a):
    if a%2==0:
        return "EVEN"
    else:
        return "ODD"

@nonnegative
def fact(n):
    f=1
    for i in range(1,n+1):
        f=f*i
    return f to Python")

print(fact(5))
print(isEven(5))
print(fact(-5))
print(isEven(-5))
```

### Output

```
120
ODD
Value must be greater than 0
Value must be greater than 0
```

# Anonymous Functions

Python Anonymous function is a function which has no name. Therefore, we can use Python Anonymous function for a small scope of program. Normally, we define Python function using def keyword. But we define anonymous function using lambda keyword. The basic structure of an anonymous function is given below. Look closely, that while taking one or more arguments the anonymous function has only one expression. So, if we want to make a function which will calculate sum of two number. Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

## Syntax:

```
lambda arguments : expression
```

## Examples :

### Program

```
sqr=lambda a : a*a  
  
z=int(input("Enter a no."))  
  
print("Square",sqr(z))
```

### Output

```
Enter a no. 5  
Square 25
```

## Examples :

### Program

```
area=lambda r : 3.14*r*r  
  
z=int(input("Enter a no."))  
  
print("Area",area(z))
```

### Output

```
Enter a no.5  
Area 78.5
```

# Python Global, Local and Nonlocal variables

The way Python uses global and local variables is maverick. While in many or most other programming languages variables are treated as global if not declared otherwise, Python deals with variables the other way around. They are local, if not otherwise declared. The driving reason behind this approach is that global variables are generally bad practice and should be avoided. In most cases where you are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out. Like in many other program structures, Python also imposes good programming habit by design.

So when you define variables inside a function definition, they are local to this function by default. That is, anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name. In other words, the function body is the scope of such a variable, i.e. the enclosing context where this name is associated with its values.

All variables have the scope of the block, where they are declared and defined. They can only be used after the point of their declaration.

## Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

### Examples :

#### Program

```
x = "CCIT"

def func():
    print("x inside:", x)

func()
print("x outside:", x)
```

#### Output

```
x inside: CCIT
x outside: CCIT
```

In the above code, we created x as a global variable and defined a func() to print the global variable x. Finally, we call the func() which will print the value of x.

What if you want to change the value of x inside a function?

## Examples :

### Program

```
x=5

def func():
    x=x+2
    print('Value of X',x)

func()
print('Value of X',x)
```

### Output

```
UnboundLocalError: local
variable 'x' referenced
before assignment
```

The output shows an error because Python treats x as a local variable and x is also not defined inside func().

## Global Keyword

In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

### Rules of global Keyword

- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
- We use global keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect.

## Examples :

### Program

```
x=5

def func():
    global x
    x=x+2
    print('Value of X',x)

func()
print('Value of X',x)
```

### Output

```
Value of X 7
Value of X 7
```

## local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

### Examples :

#### Program

```
def func():
    x="CCIT"

func()
print('Value of X',x)
```

#### Output

```
NameError: name 'x' is not defined
```

The output shows an error because we are trying to access a local variable x in a global scope whereas the local variable only works inside func() or local scope.

Normally, we declare a variable inside the function to create a local variable.

### Examples :

#### Program

```
def func():
    x="CCIT"
    print('Value of X',x)

func()
```

#### Output

```
Value of X CCIT
```

## Nonlocal Variables

Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

Let's see an example of how a global variable is created in Python.

- The Python keyword nonlocal binds one or more variables to the outer scope.
- If the immediate outer scope does not have the same name present, it will resolve to the name in the next outer scope.
- If none of the outer scopes have the same name present, it will resolve to global scope.
- In the absence of nonlocal qualification, any variable is bound to its local scope.

- Assume the program execution is at the inner most scope ('n' levels inside from a function call), the global keyword makes the name to be directly bound to global scope, while the nonlocal keyword makes the name to be bound to next outer scope at which the name is available.
- A formal parameter of a function cannot be defined as nonlocal inside that function.

### Examples :

#### Program

```
def func():
    x="CCIT"
    def inner():
        x="Amravati"
        print("Value of X is ",x)
    inner()
    print("Value of X is ",x)

func()
```

#### Output

```
Value of X Amravati
Value of X CCIT
```

We use nonlocal keywords to create nonlocal variables.

### Examples :

#### Program

```
def func():
    x="CCIT"
    def inner():
        nonlocal x
        x="Amravati"
        print("Value of X is ",x)
    inner()
    print("Value of X is ",x)

func()
```

#### Output

```
Value of X Amravati
Value of X Amravati
```

In the above code, there is a nested inner() function. We use nonlocal keywords to create a nonlocal variable. The inner() function is defined in the scope of another function func().

# Python Closures

In programming languages, a closure, also lexical closure or function closure, is a technique for implementing lexically scoped name binding in a language with first-class functions. Operationally, a closure is a record storing a function together with an environment. The environment is a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created. Unlike a plain function, a closure allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

## Examples :

### Program

```
def func():
    x="CCIT"
    def inner():
        print(x)
    return inner()

func()
```

### Output

CCIT

We can see that the nested inner() function was able to access the non-local x variable of the enclosing function.

## Defining a Closure Function

In the example above, what would happen if the last line of the function func() returned the inner() function instead of calling it? This means the function was defined as follows:

## Examples :

### Program

```
def func():
    x="CCIT"
    def inner():
        print(x)
    return inner

func_msg=func()
func_msg()
```

### Output

CCIT

The func() function was called with the string "CCIT" and the returned function was bound to the name another. On calling func\_msg(), the message was still remembered although we had already finished executing the func() function.

This technique by which some data (CCIT in this case) gets attached to the code is called closure in Python.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

## Closure Function with argument

The outer function accept one argument. argument value stored in nonlocal variable x. x variable is read only it cannot be modify.

### Examples :

#### Program

```
def func(msg):
    x=msg
    def inner():
        print(x)
    return inner

fun_msg=func("CCIT")
fun_msg()
```

#### Output

CCIT

### Examples :

#### Program

```
def num(a):
    x=a
    def inner(b):
        c=x*b
        print("Result is",c)
    return inner

square=num(2)
square(2)
square(5)

cube=num(3)
cube(2)
cube(5)
```

#### Output

**Result is 4**  
**Result is 10**  
**Result is 6**  
**Result is 15**

# Object- Oriented programming

# Object-Oriented Programming

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Another common programming paradigm is procedural programming which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of the object-oriented programming paradigm, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

Python is a multi-paradigm programming language. Meaning, it supports different programming approach. One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

# Class

A class is a blue print of an object. We can say it is generic description of an Object. A class is a user defined data type where we can group data and its related functions together. Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal class to track properties about the Animal like the name and age.

## Syntax:

```
class <classname>:
    Data members
    -----
    Member Functions
    -----
```

## Data members:

It indicates information about the object or current state of object. Data members must be created in setter methods or constructors. Data members are automatically created when value is assigned to them in member functions. All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph).

## Syntax:

```
self.Data Members= value
```

## Member functions:

It indicates the operations that we perform on the object. The self-parameter is a reference to the object itself, and is used to access members that belongs to the object. It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class.

## Syntax:

```
def <member-function>(self,parameters...):
    Statements
    -----
```

# Objects

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated. It is a variable in which we can store ID of an Object. Each Object has a unique ID. It will create a new object of specified class and will return ID of object which we can assign into reference variable.

## Syntax:

```
Object = ClassName()
```

## Examples :

### Program

```
class Rectangle:

    def setDimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)

    def perimeter(self):
        p=2*(self.length+self.breadth)
        print("Perimeter  is ",p)

a=Rectangle()
a.setDimension(12,14)
a.area()
a.perimeter()
```

### Output

```
Area is 168
Perimeter  is 52
```

**Examples :**

**Design a class Circle containing data member radius and member functions area, circumference and setradius.**

**Program**

```
class circle:
    def setradius(self,n):
        self.r=n
    def area(self):
        a=3.14*self.r**2
        print("Area is ",a)
    def circumference(self):
        c=2*3.14*self.r
        print("Circumference is ",c)

a=circle()
a.setradius(5)
a.area()
a.circumference()
```

**Output**

```
Area is 78.5
Circumference is 31.4000002
```

**Examples :**

**Design a class Worker containing data member wages and wdays and member functions payment and setData.**

**Program**

```
class worker:
    def setData(self,m,n):
        self.wages=m
        self.wdays=n
    def payment(self):
        p=self.wages*self.wdays
        print("Payment is ",p)

a=worker()
a.setData(500,5)
a.payment()
```

**Output**

```
Payment is 2500
```

# Constructors

A constructor is a special type of method (function) which is used to initialize the object of the class. Constructor are automatically invoked when object of class is created. Python class constructor is the first piece of code to be executed when you create a new object of a class.

Primarily, the constructor can be used to put values in the member variables. You may also print messages in the constructor to be confirmed whether the object has been created. The constructor method starts with `def __init__`. Afterward, the first parameter must be '`self`' , as it passes a reference to the instance of the class itself. When we define a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

## Syntax:

```
def __init__(self,parameters...):
    Statement
    -----
```

## Examples :

**Design a class Circle containing data member radius and member functions area, circumference and constructors.**

### Program

```
class circle:
    def __init__(self,n):
        self.r=n
    def area(self):
        a=3.14*self.r**2
        print("Area is ",a)
    def circumference(self):
        c=2*3.14*self.r
        print("Circumference is ",c)

a=circle(5)
a.area()
a.circumference()
```

### Output

```
Area is 78.5
Circumference is 31.4000002
```

## Examples :

**Design a class Rectangle containing data member length and breadth member functions area, perimeter and constructors.**

Program

```
class Rectangle:  
    def __init__(self,x,y):  
        self.length=x  
        self.breadth=y  
  
    def area(self):  
        a=self.length*self.breadth  
        print("Area is ",a)  
  
    def perimeter(self):  
        p=2*(self.length+self.breadth)  
        print("Perimeter is ",p)  
  
  
a=Rectangle(12,14)  
a.area()  
a.perimeter()  
  
  
b=Rectangle(42.4,21.2)  
b.area()  
b.perimeter()
```

Output

```
Area is 168  
Perimeter is 52  
Area is 898.88  
Perimeter is 127.1999999
```

# Constructors with default arguments

A constructor is a special type of method (function) which is used to initialize the object of the class. Constructor are automatically invoked when object of class is created. While create a new object if object is not initialized with any value then object should be initialized with we default value.to do this we use default constructor.

## Syntax:

```
def __init__(self,parameters=<default-value>):
    Statement
    -----
```

## Examples :

**Design a class Rectangle containing data member length and breadth member functions area, perimeter and constructors.**

### Program

```
class Time:
    def __init__(self,h=0,m=0):
        self.hours=h
        self.mins=m
    def display(self):
        print(self.hours,":",self.mins)

a=Time(2,45)
b=Time()
a.display()
b.display()
```

### Output

```
2 : 45
2 : 0
```

# Destructor

Destructors are called when the object of a class is destroyed. The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted. Destructor is a special method of class which is automatically called when object is to be deleted. This method might be used to clean up any resources used by Object.

## Syntax:

```
def __del__(self):
    Statement
    -----
```

## Examples :

### Program

```
class Time:
    def __init__(self,h,m):
        self.hours=h
        self.mins=m
    def display(self):
        print(self.hours,":",self.mins)
    def __del__(self):
        print("Object is Destroy")

a=Time(2,45)
a.display()
del a
```

### Output

```
2 : 45
Object is Destroy
```

# Class Variables

There is another type of variable you can use with classes, called a class variable, which is applied to all new instances of the class that haven't been created yet. Class variables inside a class don't have any tie-in to self, because the keyword self always refers to the specific object being created at the moment. Only one copy of class Variable is created for entire class. Class variables are shared by all objects. All variables which are assigned a value in class declaration are class variables. Instance or non-class variables are different for different objects (every object has a copy of it).

## Syntax:

```
class <class-name>:  
    <variable-name>=value
```

## Accessing Class Variables:

Class variable can be access in non-class methods or outside class.

## Syntax:

```
<class-name>.<variable-name>
```

## Examples :

**Design a class Account data members accno, balance, Irate. Irate must be shared by all objects.**

### Program

```
class Account:
    Irate=10.25
    def setData(self,an,b1):
        self.accno=an
        self.bal=b1
    def interest(self,n):
        si=self.bal*Account.Irate*n/100
        print("Simple Interest is ",si)

print("Interest Rate :",Account.Irate)
a=Account( )
b=Account( )
a.setData(4117,5000)
b.setData(3013,25000)
a.interest(3)
b.interest(2)
Account.Irate=11.75
print("Interest Rate :",Account.Irate)
```

### Output

```
Interest Rate : 10.25
Simple Interest is 1537.5
Simple Interest is 2562.5
Interest Rate : 11.75
```

## Examples :

Design a class to maintain count of all persons type of objects.

### Program

```
class Person:  
    count=0  
  
    def __init__(self,name):  
        self.name=name  
        Person.count=Person.count+1  
  
    def showdata(self):  
        print("name is "+self.name)  
  
  
print("Total persons:",Person.count)  
a=Person("Amit")  
b=Person("Raj")  
print("Total persons:",Person.count)  
c=Person("Mona")  
print("Total persons:",Person.count)  
a.showdata()  
b.showdata()  
c.showdata()
```

### Output

```
Total persons: 0  
Total persons: 2  
Total persons: 3  
name is Amit  
name is Raj  
name is Mona
```

# Class methods

A class method receives the class as implicit first argument(`cls`), just like an instance method receives the instance reference(`self`). As the name implies, a class method is a method that is associated with the class as a whole, not specific instances of the class. In other words, class methods are similar in scope to class variables in that they apply to the whole class and not just individual instances of the class. As with class variables, you don't need the `self`-keyword with class methods, because that keyword always refers to the specific object being created at the moment, not to all objects created by the class. So for starters, if you want a method to do something to the class as a whole, don't use def name (`self`) because the `self` immediately ties the method to one object. A class method can access only class members and not object members.

## Syntax:

```
@classmethod  
def <member-function>(cls,parameters...):  
    Statements  
    -----
```

## Accessing Class method:

Such methods can be called without creating object. Class method can be accessed in non-class methods or outside class.

## Syntax:

```
<classname>.MethodName(arguments...)
```

## Examples :

Design a class to maintain count of all persons type of objects.

Program

```
class Person:  
    __count=0  
  
    def __init__(self,name):  
        self.name=name  
        Person.__count=Person.__count+1  
  
    def showdata(self):  
        print("name is "+self.name)  
  
    @classmethod  
    def showcount(cls):  
        print("Total persons",cls.__count)  
  
  
Person.showcount()  
a=Person("Amit")  
b=Person("Raj")  
Person.showcount()  
c=Person("Mona")  
Person.showcount()  
a.showdata()
```

Output

```
Total persons: 0  
Total persons: 2  
Total persons: 3  
name is Amit  
name is Raj  
name is Mona
```

# Static Methods

A static method can't access or modify class or object state. A static method is used just to group related functions of class i.e. like a namespace. Anyway, underneath that `@staticmethod` line you define your static method the same as any other method, but you don't use `self` and you don't use `cls`. Because a static method isn't strictly tied to a class or object, except to the extent you want to keep it there for organizing your code. A static method does not receive an implicit first argument.

## Syntax:

```
@staticmethod
def <member-function>(parameters...):
    Statements
-----
```

## Accessing Class method:

Such methods can be called without creating object. It is similar to class methods i.e.

## Syntax:

```
<classname>.MethodName(arguments...)
```

## Examples :

### Program

```
class MyLib:

    @staticmethod
    def fact(n):
        f=1
        for i in range(1,n+1):
            f=f*i
        return f

    @staticmethod
    def sum(n):
        s=0
        for i in range(1,n+1):
            s=s+i
        return s

print("Factorial is ",MyLib.fact(5))
print("Sum is ",MyLib.sum(5))
```

### Output

```
Factorial is 120
Sum is 15
```

## Examples :

### Program

```
class mydata:  
    @staticmethod  
    def max(a,b):  
        if a>b:  
            return a  
        else:  
            return b  
  
    @staticmethod  
    def min(a,b):  
        if a<b:  
            return a  
        else:  
            return b  
  
print(mydata.max(4117,9494))  
print(mydata.min(4117,9494))
```

### Output

```
9494  
4117
```

# Passing Object as Argument

Whenever an object is pass as argument to function then its reference is pass .So the operations performed by function by using that reference are reflected back on original object.

## Syntax:

```
def <member-function>(<object>):  
    Statements  
    -----
```

## Examples :

### Program

```
class Time:  
    def __init__(self,h,m):  
        self.hours=h  
        self.mins=m  
    def display(self):  
        print(self.hours,":",self.mins)  
  
def increment(t):  
    t.hours=t.hours+1  
  
a=Time(2,45)  
a.display()  
increment(a)  
a.display()
```

### Output

```
2 : 45  
3 : 45
```

## Examples :

### Program

```
class Time:  
    def __init__(self,h,m):  
        self.__hours=h  
        self.__mins=m  
    def display(self):  
        print(self.__hours,":",self.__mins)  
  
@classmethod  
def add(cls,x,y):  
    z=Time(x.__hours+y.__hours,x.__mins+y.__mins)  
    if z.__mins>=60:  
        z.__mins-=60  
        z.__hours+=1  
    return z  
  
a=Time(2,45)  
a.display()  
b=Time(1,30)  
b.display()  
c=Time.add(a,b)  
c.display()
```

### Output

```
2 : 45  
1 : 30  
4 : 15
```

# Special Method

In Python, special methods are a set of predefined methods you can use to enrich your classes. They are easy to recognize because they start and end with double underscores, for example `_init_` or `_str_`. Class functions that begins with double underscore `_` are called special functions in Python. This is because, well, they are not ordinary.

The `_init_()` function we defined above, is one of them. It gets called every time we create a new object of that class. This elegant design is known as the Python data model and lets developers tap into rich language features like sequences, iteration, operator overloading, attribute access, etc.

## Object Representation:

It's common practice in Python to provide a string representation of your object for the consumer of your class (a bit like API documentation.) There are two ways to do this using Object Representation methods

### `__repr__()`:

Called by the `repr()` built-in function to compute the "official" string representation of an object. If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object.

### `__str__()`:

Called by `str(object)` and the built-in functions `format()` and `print()` to compute the "informal" or nicely printable string representation of an object. The return value must be a string object.

## Examples :

### Program

```
class student:
    def __init__(self,r,n):
        self.rollno=r
        self.name=n

    def __str__(self):
        msg=f'''Rollno: {self.rollno}
                \nName: {self.name}'''
        return msg

a=student(4177,"Amit Jain")
print(a)
```

### Output

```
Rollno: 4177
Name: Amit Jain
```

# Operator Overloading

Python Operator overloading enables us to use mathematical, logical and bitwise operators on python objects just like any primitive data type. Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings. This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

Redefining operators for user defined data type is called operator overloading. This can be achieved by defining a special operator function. Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. We can overload all existing operators but we can't create a new operator.

## Mathematical Operators:

<b>Operator</b>	<b>Special Method</b>	<b>Description</b>
+	<code>__add__(self, object)</code>	Addition
-	<code>__sub__(self, object)</code>	Subtraction
*	<code>__mul__(self, object)</code>	Multiplication
**	<code>__pow__(self, object)</code>	Exponentiation
/	<code>__truediv__(self, object)</code>	Division
//	<code>__floordiv__(self, object)</code>	Integer Division
%	<code>__mod__(self, object)</code>	Modulus

## Examples :

### Design a class Time to overload + operator.

Program

```
class Time:

    def __init__(self,h,m):
        self.H=h
        self.M=m

    def display(self):
        print(self.H,":",self.M)

    def __add__(self,other):
        z=Time(self.H+other.H,self.M+other.M)
        if z.mins>=60:
            z.mins=z.mins-60
            z.hours=z.hours+1
        return z

a=Time(2,30)
b=Time(1,50)
a.display()
b.display()
c=a+b
c.display()c.showdata()
c.showdata()
```

Output

```
2 : 30
1 : 50
4 : 20
```

## Examples :

**Design a class person data members name , age and overload + and – operator to add and subtract age.**

Program

```
class Person:

    def __init__(self,nm,ag):
        self.name=nm
        self.age=ag

    def display(self):
        print("Age of",self.name,"is",self.age)

    def __add__(p,n):
        z=Person(p.name,p.age+n)
        return z

    def __sub__(p,n):
        z=Person(p.name,p.age-n)
        return z

a=Person("Amit",23)
a.display()
a=a+2
a.display()
a=a-5
a.display()
```

Output

```
Age of Amit is 23
Age of Amit is 25
Age of Amit is 20
```

## Overloading Comparison Operators:

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well. Suppose, we wanted to implement the less than symbol < symbol in our Point class. Let us compare the magnitude of these points from the origin and return the result for this purpose.

Operator	Special Method	Description
<code>==</code>	<code>__eq__(self, object)</code>	Equal to
<code>!=</code>	<code>__ne__(self, object)</code>	Not equal to
<code>&gt;</code>	<code>__gt__(self, object)</code>	Greater than
<code>&gt;=</code>	<code>__ge__(self, object)</code>	Greater than or equal to
<code>&lt;</code>	<code>__lt__(self, object)</code>	Less than
<code>&lt;=</code>	<code>__le__(self, object)</code>	Less than or equal to
<code>in</code>	<code>__contains__(self, value)</code>	Membership operator

## Examples :

### Design a class Time and overload == operator

Program

```
class Time:

    def __init__(self,h,m):
        self.H=h
        self.M=m

    def display(self):
        print(self.H,":",self.M)

    def __eq__(self,other):
        if self.H==other.H and self.M==other.M:
            return True
        else:
            return False

a=Time(2,30)
b=Time(2,30 )
a.display()
b.display()
if a==b:
    print("same")
else:
    print("different")
```

Output

```
2 : 30
2 : 30
same
```

## Examples :

Design a class Time and overload > operator to compare 2 time type of objects.

Program

```
class Time:

    def __init__(self,h,m):
        self.H=h
        self.M=m

    def display(self):
        print(self.H,":",self.M)

    def __gt__(self,other):
        m1=self.H*60+self.M*60
        m2=other.H*60+other.M*60
        if m1>m2:
            return True
        else:
            return False

a=Time(4,30)
b=Time(2,30)
a.display()
b.display()
if a>b:
    print("a is Greater")
else:
    print("a is not Greater")
```

Output

```
4 : 30
2 : 30
a is Greater
```

## Overloading length and index Operators:

Python does not limit operator overloading to arithmetic operators only. When you're calling len() on an object, Python handles the call as obj.\_\_len\_\_(). When you use the [] operator on an iterable to obtain the value at an index, Python handles it as itr.\_\_getitem\_\_(index)

Operator	Special Method	Description
len()	__len__(self, object)	Length
[]	__getitem__(self, object)	index

### Examples :

Design a class order and overload len operator to find out total number of items in cart.

Program

```
class order:

    def __init__(self,crt,cst):
        self.cart=list(crt)
        self.customer=cst

    def __len__(self):
        c=0
        for i in self.cart:
            c+=1
        return c

a=order(['Apple','Orange','Mango'],"Amit Jain")

l=len(a)

print("Total item in cart are",l)
```

Output

```
Total item in cart  
are 3
```

## Examples :

Design a class order and overload [ ] operator to find out value at index.

Program

```
class order:

    def __init__(self,crt,cst):
        self.cart=list(crt)
        self.customer=cst

    def __getitem__(self,index):
        c=0
        for i in self.cart:
            if c==index:
                return i
            c+=1
        return "invalid index"

a=order(['Apple','Orange','Mango'],"Amit Jain")

print("Item at index 2 is",a[2])
```

Output

```
Item at index 2 is
Mango
```

# Inheritance

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more. Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class.

Inheritance allows programmers to create classes that are built upon existing classes, and this makes it possible that a class created through inheritance inherits the attributes and methods of the parent class. This means that inheritance supports code reusability. The methods or generally speaking the software inherited by a subclass is considered to be reused in the subclass.

The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. Derived class inherits features from the base class, adding new features to it. This results into re-usability of code. It is used to derive a new class from existing class. The derived class inherits all features of base class. A new class can be derived by providing list of base class names from which we want to derive. Every class in Python is derived from the class object. If base class name is not provided then that class is derived from class object.

## Syntax:

```
class DerivedClassName(BaseClassName):
```

```
    statement
```

```
    -----
```

## Examples :

**Design a class circle : data members r , member functions area( ) , setradius( )  
Then derive a new class xcircle from circle : memberfunction circumference( ).**

### Program

```
class circle:
    def setradius(self,r):
        self.r=r
    def area(self):
        a=3.14*self.r**2
        print("Area is ",a)

class xcircle(circle):
    def circumference(self):
        c=2*3.14*self.r
        print("circumference is ",c)

a=circle()
a.setradius(5)
a.area()

b=xcircle()
b.setradius(1)
b.area()
b.circumference()
```

### Output

```
Area is  78.5
Area is  3.14
circumference is  6.28
```

## Examples :

**Design a class Account : data members: accno, balance and member function open(..) , deposit( . ) , withdraw( . ), showBalance( ) Then derive a new class SAaccount from account : member function addInterest( .. )**

### Program

```
class Account:
    def open(self,an,b1):
        self.accno=an
        self.bal=b1
    def deposit(self,amt):
        self.bal=self.bal+amt
    def withdraw(self,amt):
        self.bal=self.bal-amt
    def showBalance(self):
        print("AccNo is ",self.accno)
        print("Bal is ",self.bal)

class SAccount(Account):
    def addInterest(self,r,n):
        si=self.bal*r*n/100
        self.bal=self.bal+si

a=Account()
a.open(4117,25000)
a.deposit(3000)
a.withdraw(8000)
a.showBalance()

b=SAccount()
b.open(3012,10000)
b.deposit(40000)
b.showBalance()
b.addInterest(10.25,3)
b.showBalance()
```

### Output

```
AccNo is 4117
Bal is 20000
AccNo is 3012
Bal is 50000
AccNo is 3012
Bal is 65375.0
```

# Inheritance Method Overriding

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more. Redefining a base class method in derived class is call as method overriding. The override method must have same name as base class method. This override method will be call for derived class object instead of base class inherited method.

## Examples :

**class Box**

**Data members**

Length

Breadth

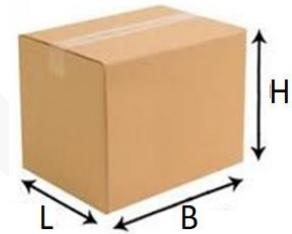
Height

**Member functions**

Setdimension(x,y,z)

Volume() //v=L\*B\*H

surfaceArea() //A=2\*L\*H+2\*B\*H+2\*L\*B



Then derive a new class OpenBox from class Box.

**Class OpenBox(Box):**

**Member functions**

SurfaceArea( ) //A=2\*L\*H+2\*B\*H+L\*B

## Examples :

### Program

```
class Box:

    def setdata(self,x,y,z):
        self.L=x
        self.B=y
        self.H=z

    def volume(self):
        v=self.L*self.B*self.H
        print("Volume is ",v)

    def surfacearea(self):
        a=2*self.L*self.B+2*self.B*self.H+2
        *self.L*self.B
        print("SurfaceArea is ",a)

class OpenBox(Box):

    def surfacearea(self):
        a=self.L*self.B+2*self.B*self.H+2*s
        elf.L*self.B
        print("SurfaceArea is ",a)

a=Box()
a.setdata(2,2,2)
a.volume()
a.surfacearea()

b=OpenBox()
b.setdata(1,1,1)
b.volume()
b.surfacearea()
```

### Output

```
Volume is  8
SurfaceArea is  24
Volume is  1
SurfaceArea is  5
```

## Examples :

**Design a class Account : data members: accno, balance and member function open( . . ) , deposit( . ) , withdraw( . ), showBalance( ) Then derive a new class CAaccount from Account which will charge 1 rupee for each transaction. i.e override deposit( . ) and withdraw( . )**

### Program

```
class Account:
    def open(self,an,bl):
        self.accno=an
        self.balance=bl
    def deposit(self,amt):
        self.balance=self.balance+amt
    def withdraw(self,amt):
        self.balance=self.balance-amt
    def showBalance(self):
        print("AccNo is ",self.accno)
        print("Balance is ",self.balance)

class CAaccount(Account):
    def deposit(self,amt):
        self.balance=self.balance+amt
        self.balance=self.balance -1
    def withdraw(self,amt):
        self.balance=self.balance-amt
        self.balance=self.balance -1

a=Account()
a.open(4117,25000)
a.deposit(5000)
a.withdraw(2000)
a.showBalance()

b=CAaccount()
b.open(3012,10000)
b.deposit(5000)
b.withdraw(2000)
b.withdraw(3000)
b.showBalance()
```

### Output

```
AccNo is 4117
Balance is 28000
AccNo is 3012
Balance is 9997
```

# Calling base class overrided method in derived class

If required a base class function can be call from derived class. The first argument must be self.

**Syntax:**

```
BaseClassName.methodName(self,arg1,arg2, . . )
```

**Examples :**

**Design a class Account : data members: accno, balance and member function open( . . ), deposit( . ), withdraw( . ), showBalance( )**

**Then derive a new class CAaccount from Account which will charge 1 rupee for each transaction. i.e override deposit( . ) and withdraw( . )**

## Program

```

class Account:
    def open(self,an,bl):
        self.accno=an
        self.balance=bl
    def deposit(self,amt):
        self.balance=self.balance+amt
    def withdraw(self,amt):
        self.balance=self.balance-amt
    def showBalance(self):
        print("AccNo is ",self.accno)
        print("Balance is ",self.balance)

class CAccount(Account):
    def deposit(self,amt):
        Account.deposit(self,amt)
        self.balance=self.balance -1
    def withdraw(self,amt):
        Account.withdraw(self,amt)
        self.balance=self.balance -1

a=Account()
a.open(4117,25000)
a.deposit(5000)
a.withdraw(2000)
a.showBalance()

b=CAccount()
b.open(3012,10000)
b.deposit(5000)
b.withdraw(2000)
b.withdraw(3000)
b.showBalance()

```

## Output

```

AccNo is 4117
Balance is 28000
AccNo is 3012
Balance is 9997

```

# Super( ) function

Python super() function allows us to refer to the parent class explicitly. It's useful in case of inheritance where we want to call super class functions. The super() builtin returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.

The super() builtin returns a proxy object, a substitute object that can call methods of the base class via delegation. This is called indirection (ability to reference base object with super()) Since the indirection is computed at the runtime, we can use different base classes at different times (if we need to).

Note: first argument self is not required.

## Syntax:

```
super( ).baseClassFnName(arg1,arg2, . . . )
```

## Examples :

### Program

```
class Person:
    def setdata(self,name):
        self.name=name
    def showdata(self):
        print("Name is ",self.name)

class Student(Person):
    def setdata(self,rollno,name):
        self.rollno=rollno
        super().setdata(name)
    def showdata(self):
        print("RollNo is",self.rollno)
        super().showdata()

a=Person()
a.setdata("Amit jain")
a.showdata()

b=Student()
b.setdata(4117,"Gopal Pandey")
b.showdata()
```

### Output

```
Name is Amit jain
RollNo is 4117
Name is Gopal Pandey
```

## Examples :

### Program

```
class Person:

    def __init__(self, name):
        self.name = name

    def showdata(self):
        print("Name is ", self.name)

class Student(Person):

    def __init__(self, rollno, name):
        self.rollno = rollno
        super().__init__(name)

    def showdata(self):
        print("RollNo is", self.rollno)
        super().showdata()

a = Person("Amit jain")
a.showdata()
b = Student(4117, "Gopal Pandey")
b.showdata()
```

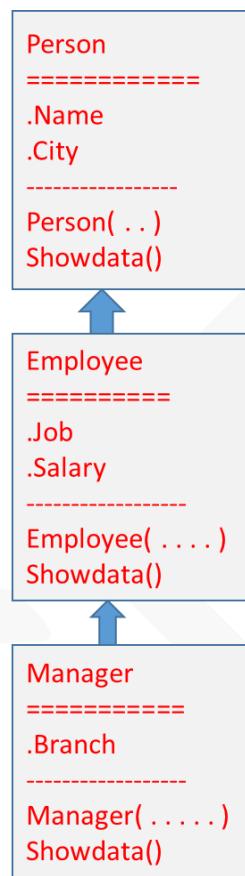
### Output

```
Name is Amit jain
RollNo is 4117
Name is Gopal Pandey
```

## Examples :

**Design 3 classes Person , Employee and Manager from given Inheritance Diagram**

Program



```

class Person:
    def __init__(self,nm,ct):
        self.name=nm
        self.city=ct
    def showdata(self):
        print("Name is ",self.name)
        print("City is ",self.city)
    
```

## Program

```

class Employee(Person):
    def __init__(self,nm,ct,jb,sl):
        super().__init__(nm,ct)
        self.job=jb
        self.salary=sl
    def showdata(self):
        super().showdata()
        print("Job is ",self.job)
        print("Salary is ",self.salary)

class Manager(Employee):
    def __init__(self,nm,ct,jb,sl,br):
        super().__init__(nm,ct,jb,sl)
        self.branch=br
    def showdata(self):
        super().showdata()
        print("Branch is ",self.branch)

a=Person("Amit jain","Amrvati")
a.showdata()

b=Employee("Raj.Joshi","Nagpur","Clerk",25000)
b.showdata()

c=Manager("M.Rao","Mumbai","Sr.Manager",45500,"Camp-Amt")
c.showdata()

```

## Output

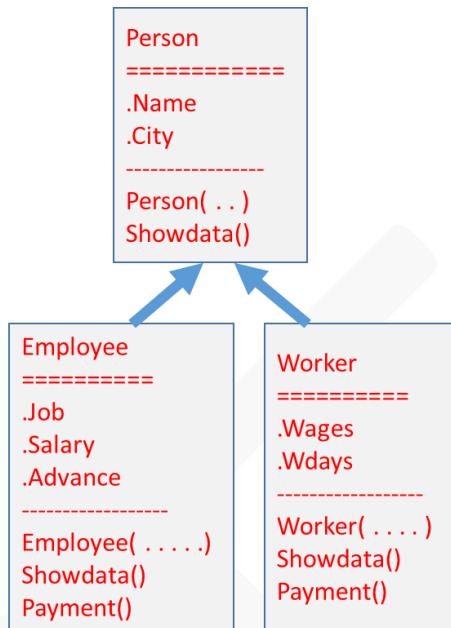
```

Name is Amit jain
City is Amrvati
Name is Raj.Joshi
City is Nagpur
Job is Clerk
Salary is 25000
Name is M.Rao
City is Mumbai
Job is Sr.Manager
Salary is 45500
Branch is Camp-Amt

```

## Examples :

**Design 3 classes Person Employee and worker from given Inheritance diagram**



### Program

```

class Person:
    def __init__(self,nm,ct):
        self.name=nm
        self.city=ct
    def showdata(self):
        print("Name is ",self.name)
        print("City is ",self.city)
  
```

### Output

```

Name is Raj.Joshi
City is Nagpur
Job is Clerk
Salary is 25000
Advance is 5000
payment is 20000
Name is Raja.Kumar
City is Raipur
Wages is 500
Wdays is 25
payment is 12500
  
```

## Program

```
class Employee(Person):
    def __init__(self,nm,ct,jb,sl,ad):
        super().__init__(nm,ct)
        self.job=jb
        self.salary=sl
        self.advance=ad
    def showdata(self):
        super().showdata()
        print("Job is ",self.job)
        print("Salary is ",self.salary)
        print("Advance is ",self.advance)
    def payment(self):
        p=self.salary-self.advance
        print("payment is ",p)

class Worker(Person):
    def __init__(self,nm,ct,wg,wd):
        super().__init__(nm,ct)
        self.wages=wg
        self.wdays=wd
    def showdata(self):
        super().showdata()
        print("Wages is ",self.wages)
        print("Wdays is ",self.wdays)
    def payment(self):
        p=self.wages*self.wdays
        print("payment is ",p)

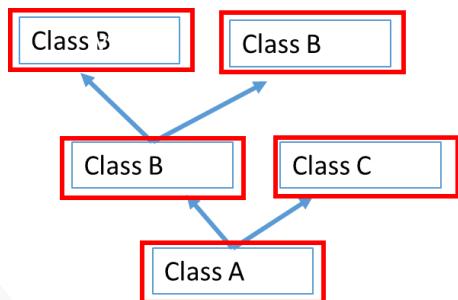
b=Employee("Raj.Joshi","Nagpur","Clerk",25000,5000)
b.showdata()
b.payment()

c=Worker("Raja.Kumar","Raipur",500,25)
c.showdata()
c.payment()
```

# Method Resolution Order

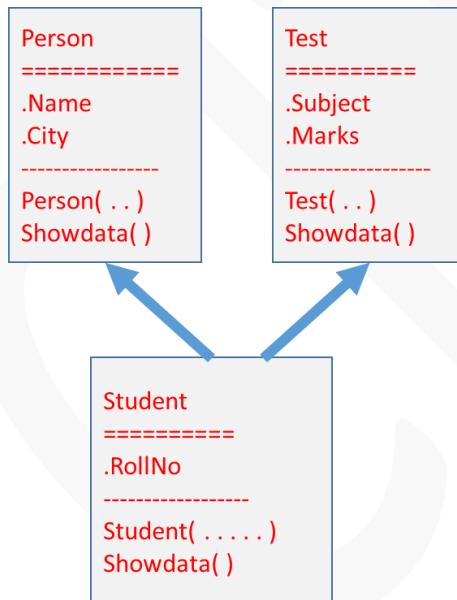
In the multiple inheritance scenario, any specified attribute is searched first in the current class.

If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.



## Examples :

**Design 3 classes Person Test and Student from given Inheritance diagram.**



## Program

```

class Person:

    def __init__(self, name, city):
        self.name = name
        self.city = city

    def showdata(self):
        print("Name is ", self.name)
        print("City is ", self.city)
  
```

## Program

```
class Test:

    def __init__(self,subject,marks):
        self.subject=subject
        self.marks=marks

    def showdata(self):
        print("Subject is ",self.subject)
        print("Marks is ",self.marks)

class Student(Person,Test):

    def __init__(self,rollno,name,city,subject,marks):
        self.rollno=rollno
        Person.__init__(self,name,city)
        Test.__init__(self,subject,marks)

    def showdata(self):
        print("RollNo is",self.rollno)
        Person.showdata(self)
        Test.showdata(self)

a=Student(4117,"G.Pandey","Amt","CET",55);
a.showdata()
```

## Output

```
RollNo is 4117
Name is G.Pandey
City is Amt
Subject is CET
Marks is 55
```

# Python public,private,protected Access Modifiers

Classical object-oriented languages, such as C++ and Java, control the access to class resources by public, private and protected keywords. Private members of a class are denied access from the environment outside the class. They can be handled only from within the class.

Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behaviour of protected and private access specifiers.

## Public Access Modifiers

All members in a Python class are public by default. Any member can be accessed from outside the class environment.

### Example:

#### Program

```
class employee:
    def __init__(self, name, sal):
        self.name = name
        self.salary = sal

    emp = employee("Amit Jain", 2000)
    print(emp.salary)
    emp.salary = 3000
    print(emp.salary)
```

#### Output

```
2000
3000
```

## Protected Access Modifiers

Python's convention to make an instance variable protected is to add a prefix `_` (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.

In fact, this doesn't prevent instance variables from accessing or modifying the instance.

### Example:

#### Program

```
class employee:
    def __init__(self, name, sal):
        self._name=name
        self._salary=sal

emp=employee("Amit Jain",2000)
print(emp._salary)
emp._salary=3000
print(emp._salary)
```

#### Output

```
2000
3000
```

## Private Access Modifiers

Similarly, a double underscore `__` prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an `AttributeError`

In fact, this doesn't prevent instance variables from accessing or modifying the instance.

### Example:

#### Program

```
class employee:
    def __init__(self, name, sal):
        self.__name=name
        self.__salary=sal

emp=employee("Amit Jain",2000)
print(emp.__salary)
emp.__salary=3000
print(emp.__salary)
```

#### Output

```
AttributeError: 'employee' object has no attribute '__salary'
```

## Setter methods

Generally data members are kept private. So we can not access such private data members outside the object. setter functions are used to set values for private data members of object. Generally their name starts with the word set.

### Example:

#### Program

```
class employee:  
    def __init__(self,name,sal):  
        self.__name=name  
        self.__salary=sal  
  
    def get_salary(self):  
        return self.__salary  
  
    def set_salary(self,sal):  
        self.__salary=sal  
  
    def display(self):  
        print("Name:",self.__name)  
        print("Salary:",self.__salary)  
  
emp=employee("Amit Jain",2000)  
emp.display()  
  
emp.set_salary(3000)  
print("Updated Salary:",emp.get_salary())
```

#### Output

```
Name: Amit Jain  
Salary: 2000  
Updated Salary: 3000
```

# Python property function

Traditional object-oriented languages like Java and C# use properties in a class to encapsulate data. Property includes the getter and setter method to access encapsulated data. A class in Python can also include properties by using the property() function.

In Python, property() is a built-in function that creates and returns a property object. The main purpose of any decorator is to change your class methods or attributes in such a way so that the user of your class no need to make any change in their code.

The property() method takes the get, set and delete methods as arguments and returns an object of the property class.

## Syntax:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

The property() takes four optional parameters:

- fget (optional) - Function for getting the attribute value. Defaults to None.
- fset (optional) - Function for setting the attribute value. Defaults to None.
- fdel (optional) - Function for deleting the attribute value. Defaults to None.
- doc (optional) - A string that contains the documentation (docstring) for the attribute. Defaults to None.

property() function returns the property attribute from the given getter, setter, and deleter.

- If no arguments are given, property() returns a base property attribute that doesn't contain any getter, setter or deleter.
- If doc isn't provided, property() takes the docstring of the getter function.

**Example:****Program**

```
class employee:
    def __init__(self, name, sal):
        self.__name=name
        self.__salary=sal
    def get_salary(self):
        return self.__salary
    def set_salary(self,sal):
        self.__salary=sal
    def del_salary(self):
        self.__salary=None
    def display(self):
        print("Name:",self.__name)
        print("Salary:",self.__salary)
    salary=property(get_salary, set_salary, del_salary)

emp=employee("Amit Jain",2000)
emp.display()
emp.salary=3000
print("Update Salary",emp.salary)
del emp.salary
emp.display()
```

**Output**

```
Name: Amit Jain
Salary: 2000
Update Salary 3000
Name: Amit Jain
Salary: None
```

Here, `__salary` is used as the private variable for storing the salary of employee.

We also set:

- a getter method `get_salary()` to get the salary of the person,
- a setter method `set_salary()` to set the salary of the person,
- a deleter method `del_salary()` to delete the name of the person.

Now, we set a new property attribute `salary` by calling the `property()` method.

As shown in the program, referencing `p.salary` internally calls `get_salary()` as getter, `set_salary()` as setter and `del_salary()` as deleter through the printed output present inside the methods.

# Python @property decorator

@property decorator allows us to define properties easily without calling the property() function manually. @property decorator is a built-in decorator in Python for the property() function. we can use the Python decorator @property to assign the getter, setter, and deleter.

## @property decorator getter

A getter - to access the value of the attribute.

### Example:

#### Program

```
class employee:
    def __init__(self, name, sal):
        self.__name = name
        self.__salary = sal

    @property
    def salary(self):
        return self.__salary

    def set_salary(self, sal):
        self.__salary = sal

    def display(self):
        print("Name:", self.__name)
        print("Salary:", self.__salary)

emp = employee("Amit Jain", 2000)
emp.display()

emp.set_salary(3000)
print("Update Salary", emp.salary)
```

#### Output

```
Name: Amit Jain
Salary: 2000
Update Salary 3000
```

### Note:

- @property - Used to indicate that we are going to define a property. Notice how this immediately improves readability because we can clearly see the purpose of this method.
- def salary(self) - The header. Notice how the getter is named exactly like the property that we are defining: salary. This is the name that we will use to access and modify the attribute outside of the class. The method only takes one formal parameter, self, which is a reference to the instance.
- return self.\_\_salary - This line is exactly what you would expect in a regular getter. The value of the private attribute is returned.

## @property decorator setter

A setter - to set the value of the attribute.

### Example:

#### Program

```
class employee:
    def __init__(self, name, sal):
        self.__name=name
        self.__salary=sal

    @property
    def salary(self):
        return self.__salary

    @salary.setter
    def salary(self, sal):
        self.__salary=sal

    def display(self):
        print("Name:", self.__name)
        print("Salary:", self.__salary)

emp=employee("Amit Jain", 2000)
emp.display()

emp.salary=3000
print("Update Salary", emp.salary)
```

#### Output

```
Name: Amit Jain
Salary: 2000
Update Salary 3000
```

### Note:

- @salary.setter - Used to indicate that this is the setter method for the salary property. Notice that we are not using @property.setter, we are using @salary.setter. The name of the property is included before .setter.
- def salary(self, sal): - The header and the list of parameters. Notice how the name of the property is used as the name of the setter. We also have a second formal parameter (sal), which is the new value that will be assigned to the salary attribute (if it is valid).

## @property decorator deleter

A deleter - to delete the instance attribute.

### Example:

#### Program

```
class employee:
    def __init__(self, name, sal):
        self.__name=name
        self.__salary=sal

    @property
    def salary(self):
        return self.__salary

    @salary.setter
    def salary(self, sal):
        self.__salary=sal

    @salary.deleter
    def salary(self):
        self.__salary=None

    def display(self):
        print("Name:", self.__name)
        print("Salary:", self.__salary)

emp=employee("Amit Jain", 2000)
emp.display()

emp.salary=3000
print("Update Salary", emp.salary)

del emp.salary
emp.display()
```

#### Output

```
Name: Amit Jain
Salary: 2000
Update Salary 3000
Name: Amit Jain
Salary: None
```

### Note:

- @salary.deleter - Used to indicate that this is the deleter method for the price property. Notice that this line is very similar to @salary.setter, but now we are defining the deleter method, so we write @salary.deleter.
- def salary(self): - The header. This method only has one formal parameter defined, self.

# Polymorphism

In programming, polymorphism means same function name being used for different types. i.e. same method can be implemented for different objects in different ways.

Python by default performs dynamic linking. i.e. which method to be call is decided at runtime depending on type of object. Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

## Examples :

### Program

```
class India():

    def capital(self):
        print("Capital:New Delhi")
    def language(self):
        print("primary language:Hindi")

class USA():

    def capital(self):
        print("Capital:Washington")
    def language(self):
        print("primary language:English")

class PAK():

    def capital(self):
        print("Capital:Karachi")
    def language(self):
        print("primary language:Urdu")

lst=[India(),USA(),PAK()]

for cty in lst:
    print()
    cty.capital()
    cty.language()
```

### Output

```
Capital:New Delhi
primary language:Hindi

Capital:Washington
primary language:English

Capital:Karachi
primary language:Urdu
```

## Examples :

### Program

```
class Parrot:  
    def fly(self):  
        print("Parrot can fly")  
    def swim(self):  
        print("Parrot can't swim")  
  
class Penguin:  
    def fly(self):  
        print("Penguin can't fly")  
    def swim(self):  
        print("Penguin can swim")  
  
def flying_test(bird):  
    bird.fly()  
  
blu = Parrot()  
peggy = Penguin()  
  
flying_test(blu)  
flying_test(peggy)
```

### Output

```
Parrot can fly  
Penguin can't fly
```

# isinstance function

It is used to check datatype of object or value. The `isinstance()` function checks if the object (first argument) is an instance or subclass of classinfo class (second argument). It returns True if the specified object is of the specified type, otherwise False.

## Syntax:

```
isinstance(object, type)
```

## Examples :

### Program

```
x = isinstance(5, int)
print(x)

y = isinstance("Hello",int)
print(y)

z = isinstance("Hello",str)
print(z)
```

### Output

```
True
False
True
```

## Examples :

Program

```
class India():
    def capital(self):
        print("Capital:New Delhi")
    def language(self):
        print("primary language:Hindi")

class USA():
    def capital(self):
        print("Capital:Washington")
    def language(self):
        print("primary language:English")

class PAK():
    def capital(self):
        print("Capital:Karachi")
    def language(self):
        print("primary language:Urdu")
    def terrorist(self):
        print("1000s of terrorist..")

lst=[India(),USA(),PAK()]
for cty in lst:
    print()
    cty.capital()
    cty.language()
    if isinstance(cty,PAK):
        cty.terrorist()
```

Output  
Capital:New Delhi

primary language:Hindi

Capital:Washington

primary language:English

Capital:Karachi

primary language:Urdu

1000s of terrorist..

# issubclass function

The `issubclass()` function checks if the object argument (first argument) is a subclass of classinfo class (second argument). Python `issubclass()` is a built-in function that returns true if a class supplied as the first argument is the subclass of another class supplied as the second argument, else it returns false.

## Syntax:

```
issubclass(object, type)
```

## Examples :

### Program

```
class Person:  
    pass  
  
class Student(Person):  
    pass  
  
x = issubclass(Student, Person)  
print(x)
```

### Output

```
True
```

# Abstract Methods and Classes

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes cannot be instantiated, and require subclasses to provide implementations for the abstract methods.

If a method is declared as abstract then the class will automatically become abstract class. We cannot create objects of abstract class. Such classes are used to achieve polymorphism. Multiple classes can be derived from an abstract class. Whenever a class is derived from abstract class it has to override all abstract methods of base class otherwise the derived class will also become an abstract class.

## Python Abstract Classes

In fact, Python on its own doesn't provide abstract classes. Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called abc. In python an abstract class must be derived from class ABC. All abstract methods of this abstract class must be defined with decorator(attribute) @abstractmethod.

### Example:

```
from abc import ABC , abstractmethod

class Shape(ABC):

    @abstractmethod

    def area(self):

        pass


a=Shape()

#will not create object of class shape as shape is an abstract class
```

## Examples :

Program

```
from abc import ABC , abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self,n):
        self.r=n
    def area(self):
        a=3.14*self.r**2
        print("Area is ",a)

class Rectangle(Shape):
    def __init__(self,m,n):
        self.l=m
        self.b=n
    def area(self):
        a=self.l*self.b
        print("Area is ",a)

p=Circle(5)
p.area()
q=Rectangle(5,10)
q.area()
```

Output

```
Area is 78.5
Area is 50
```

## Examples :

Program

```
from abc import ABC , abstractmethod

class Account(ABC):

    def __init__(self,an,b1):
        self.accno=an
        self.balance=b1

    def showBalance(self):
        print("AccNo is ",self.accno)
        print("Balance is ",self.balance)

    @abstractmethod
    def deposit(self,amt):
        pass

    @abstractmethod
    def withdraw(self,amt):
        pass

class CAccount(Account):

    def deposit(self,amt):
        self.balance=self.balance+amt-1

    def withdraw(self,amt):
        if self.balance>amt:
            self.balance=self.balance - amt -1
        else:
            print("Insufficient Balance")
```

```
class SAccount(Account):  
    def deposit(self,amt):  
        if self.balance+amt<=100000:  
            self.balance=self.balance+amt  
        else:  
            print("Cannot deposit")  
    def withdraw(self,amt):  
        if self.balance-amt>=500:  
            self.balance=self.balance - amt  
        else:  
            print("Cannot withdraw")  
  
a=CAccount(4117,25000)  
a.deposit(5000)  
a.showBalance()  
b=SAccount(2012,10000)  
b.withdraw(9700)  
b.showBalance()
```

Output

```
AccNo is 4117  
Balance is 29999  
Cannot withdraw  
AccNo is 2012  
Balance is 10000
```

# Collections

---

---

# Python String

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used. In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn more about Unicode from [here](#).

One of the most common data type of python is String. "str" is the built in string class of python. String literals can be enclosed by single or double quotes.

Python accepts single ('), double ("") and triple (""""") quotes to denote string. Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings.

String Quotes	Example
Single quotes	'Welcome to CCIT'
Double quotes	"Welcome to CCIT"
Triple quotes	<pre>'''Welcome to CCIT Amravati'''</pre> <pre>"""Welcome to CCIT Amravati"""</pre>

## String Operators:

Operator	Description
+	The + operator concatenates strings. It returns a string consisting of the operands joined together
*	Repetition - Creates new strings, concatenating multiple copies of the same string.
In	Python also provides a membership operator that can be used with strings. The in operator returns True if the first operand is contained within the second, and False

**Example:****Program**

```
a="Welcome "
b="to "
c="CCIT"

msg=a+b+c

print(msg)
```

**Output**

```
Welcome to CCIT
```

**Example:****Program**

```
a="CCIT"

msg=a*3

print(msg)
```

**Output**

```
CCITCCITCCIT
```

**Example:****Program**

```
a="Welcome to CCIT"
msg="CCIT" in a
print(msg)
```

**Output**

```
True
```

## String Indexing:

In Python, strings are ordered sequences of character data, and thus can be indexed in this way. Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets.

String indexing in Python is zero-based: the first character in the string has index 0, the next has index 1, and so on. The index of the last character will be the length of the string minus one.

String indices can also be specified with negative numbers, in which case indexing occurs from the end of the string backward

-6	-5	-4	-3	-2	-1
P	Y	T	H	O	N
0	1	2	3	4	5

Operator	Description
<b>[index]</b>	Slice - Gives the character from the given index
<b>[startindex : endindex ]</b>	Range Slice - Gives the characters from the given range

### Example:

#### Program

```
a="Welcome to CCIT"
print(a[12])
```

#### Output

C

### Example:

#### Program

```
a="Welcome to CCIT"
print(a[5:])
print(a[:10])
print(a[5:12])
```

#### Output

me to CCIT  
Welcome to  
me to C

## String Functions:

Python provides many functions that are built-in that work with string.

Function	Description
<b>len(object)</b>	Returns the length of a string
<b>str(object)</b>	Returns a string representation of an object

### Example:

#### Program

```
a="Welcome to CCIT"
l=len(a)
print("Length of String ",l)
```

#### Output

Length of String 15

## String Methods:

Python String provides many methods that are built-in that work with string.

### Case Conversion:

Methods in this group perform case conversion on the target string.

Function	Description
<b>s.capitalize()</b>	s.capitalize() returns a copy of s with the first character converted to uppercase and all other characters converted to lowercase.
<b>s.lower()</b>	s.lower() returns a copy of s with all alphabetic characters converted to lowercase.
<b>s.swapcase()</b>	s.swapcase() returns a copy of s with uppercase alphabetic characters converted to lowercase and vice versa
<b>s.title()</b>	s.title() returns a copy of s in which the first letter of each word is converted to uppercase and remaining letters are lowercase
<b>s.upper()</b>	s.upper() returns a copy of s with all alphabetic characters converted to uppercase.

**Example:****Program**

```
a="Welcome to CCIT"
print(a.capitalize())
print(a.lower())
print(a.swapcase())
print(a.title())
print(a.upper())
```

**Output**

```
#capitalize
Welcome to ccit

#lower
welcome to ccit

#swapcase
wELCOME TO ccit

#title
Welcome To Ccit

#upper
WELCOME TO CCIT
```

**Find and Replace:**

These methods provide various means of searching the target string for a specified substring.

<b>Function</b>	<b>Description</b>
<b>s.count(sub,start,end)</b>	s.count(<sub>) returns the number of non-overlapping occurrences of substring.
<b>s.endswith(suffix,start,end)</b>	s.endswith(<suffix>) returns True if s ends with the specified <suffix> and False otherwise
<b>s.find(sub,start,end)</b>	.find() to see if a Python string contains a particular substring. s.find(<sub>) returns the lowest index in s where substring <sub> is found.
<b>s.index(sub,start,end)</b>	This method is identical to .find(), except that it raises an exception if <sub> is not found rather than returning -1
<b>s.rfind(sub,start,end)</b>	s.rfind(<sub>) returns the highest index in s where substring <sub> is found
<b>s.rindex(sub,start,end)</b>	This method is identical to .rfind(), except that it raises an exception if <sub> is not found rather than returning -1
<b>s.startswith(prefix,start,end)</b>	.startswith() method, s.startswith(<suffix>) returns True if s starts with the specified <suffix> and False otherwise

**Example:****Program**

```
a="Welcome to CCIT"
print(a.count('C'))
print(a.endswith('C'))
print(a.find('C'))
print(a.index('C'))
print(a.rfind('C'))
print(a.rindex('C'))
print(a.startswith('C'))
```

**Output**

```
#count
2

#endswith
False

#find
11

#index
11

#rfind
12

#rindex
12

startswith
False
```

**Character Classification:**

Methods in this group classify a string based on the characters it contains.

<b>Function</b>	<b>Description</b>
<b>s.isalnum()</b>	s.isalnum() returns True if s is nonempty and all its characters are alphanumeric (either a letter or a number), and False otherwise.
<b>s.isalpha()</b>	s.isalpha() returns True if s is nonempty and all its characters are alphabetic, and False otherwise
<b>s.isdigit()</b>	.isdigit() Python method to check if your string is made of only digits. s.isdigit() returns True if s is nonempty and all its characters are numeric digits, and False otherwise
<b>s.isupper()</b>	s.isupper() returns True if s is nonempty and all the alphabetic characters it contains are uppercase, and False otherwise.
<b>s.islower()</b>	s.islower() returns True if s is nonempty and all the alphabetic characters it contains are lowercase, and False otherwise.
<b>s.isspace()</b>	s.isspace() returns True if s is nonempty and all characters are whitespace characters, and False otherwise.

**Example:**

Program	Output
print("CCIT".isalnum())	#isalnum True
print("CCIT".isalpha())	#isalpha True
print("CCIT".isdigit())	#isdigit False
print("CCIT".isupper())	#isupper True
print("CCIT".islower())	#islower False
print("CCIT".isspace())	#isspace False

**String Formatting:**

Methods in this group modify or enhance the format of a string.

Function	Description
<b>s.center(width,fill)</b>	s.center(<width>) returns a string consisting of s centered in a field of width <width>.
<b>s.expandtabs(tabsize=8)</b>	s.expandtabs() replaces each tab character ('\t') with spaces. By default, spaces are filled in assuming a tab stop at every eighth column
<b>s.ljust(width,fill)</b>	s.ljust(<width>) returns a string consisting of s left-justified in a field of width <width>.
<b>s.lstrip(chars)</b>	s.lstrip() returns a copy of s with any whitespace characters removed from the left end.
<b>s.replace(old,new,count)</b>	.replace() method. s.replace(<old>, <new>) returns a copy of s with all occurrences of substring <old> replaced by <new>
<b>s.rjust(width,fill)</b>	s.rjust(<width>) returns a string consisting of s right-justified in a field of width <width>.
<b>s.rstrip(chars)</b>	s.rstrip() returns a copy of s with any whitespace characters removed from the right end

<b>s.strip(chars)</b>	s.strip() is essentially equivalent to invoking s.lstrip() and s.rstrip() in succession. Without the <chars> argument, it removes leading and trailing whitespace
<b>s.zfill(width)</b>	s.zfill(<width>) returns a copy of s left-padded with '0' characters to the specified <width>

**Example:****Program**

```
a="CCIT"

print(a.center(10,"-"))

print(a.expandtabs(tabsize=8))

print(a.ljust(10,"-"))

print(a.lstrip('C'))

print(a.replace(a,"Python"))

print(a.rjust(10,"-"))

print(a.rstrip("T"))

print(a.strip("C"))

print(a.zfill(10))
```

**Output**

```
#center
---CCIT---

#expandtabs
    CCIT

#ljust
CCIT-----

#lstrip
IT

#replace
Python

#rjust
-----CCIT

#rstrip
CCI

#strip
IT

#zfill
000000CCIT
```

## Converting Between Strings and Lists:

Methods in this group convert between a string and some composite data type by either pasting objects together to make a string, or by breaking a string up into pieces.

Function	Description
<code>s.join(iterable)</code>	<code>s.join(&lt;iterable&gt;)</code> returns the string that results from concatenating the objects in <code>&lt;iterable&gt;</code> separated by <code>s</code> .
<code>s.partition(&lt;sep&gt;)</code>	<code>s.partition(&lt;sep&gt;)</code> splits <code>s</code> at the first occurrence of string <code>&lt;sep&gt;</code> . The return value is a three-part tuple consisting of: <ul style="list-style-type: none"> <li>• The portion of <code>s</code> preceding <code>&lt;sep&gt;</code></li> <li>• <code>&lt;sep&gt;</code> itself</li> <li>• The portion of <code>s</code> following <code>&lt;sep&gt;</code></li> </ul>
<code>s.rpartition(&lt;sep&gt;)</code>	<code>s.rpartition(&lt;sep&gt;)</code> functions exactly like <code>s.partition(&lt;sep&gt;)</code> , except that <code>s</code> is split at the last occurrence of <code>&lt;sep&gt;</code> instead of the first occurrence.
<code>s.rsplit(sep=None, maxsplit=-1)</code>	Without arguments, <code>s.rsplit()</code> splits <code>s</code> into substrings delimited by any sequence of whitespace and returns the substrings as a list. If <code>&lt;sep&gt;</code> is specified, it is used as the delimiter for splitting.
<code>s.split(sep=None, maxsplit=-1)</code>	<code>s.split()</code> behaves exactly like <code>s.rsplit()</code> , except that if <code>&lt;maxsplit&gt;</code> is specified, splits are counted from the left end of <code>s</code> rather than the right end.

### Example:

#### Program

```
print("*".join(['Python', 'Flask', "Django"]))

msg="Welcome to CCIT"

print(msg.partition(" "))

print(msg.rpartition(" "))

print(msg.rsplit(" "))

print(msg.split(" "))
```

#### Output

```
#join
Python*Flask*Django

#partition
('Welcome', ' ', 'to CCIT')

#rpartition
('Welcome to', ' ', 'CCIT')

#rsplit
['Welcome', 'to', 'CCIT']

#split
['Welcome', 'to', 'CCIT']
```

## String Formatting (% Operator):

Strings in Python have a unique built-in operation that can be accessed with the % operator. Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

### Syntax:

```
" %s %d %(variable1,variable2...)
```

### Example:

#### Program

```
a="CCIT"
b="Amravati"
msg="Welcome to %s %s"%(a,b)
print(msg)
```

#### Output

Welcome to CCIT Amravati

### Example:

#### Program

```
a=35
b=21
c=a+b
msg="sum is %d"%(c)
print(msg)
```

#### Output

sum is 56

## String Formatting (str.format):

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contains curly braces {} as placeholders or replacement fields which gets replaced.

### Syntax:

```
"{} {} ".format(variable1,variable2...)
```

### Example:

#### Program

```
a="CCIT"  
b="Amravati"  
msg="Welcome to {} {}".format(a,b)  
print(msg)
```

#### Output

```
Welcome to CCIT Amravati
```

### Example:

#### Program

```
a=23  
b=10  
c=a+b  
msg="Sum is {} ".format(c)  
print(msg)
```

#### Output

```
Sum is 33
```

## String Literal:

Literals can be defined as a data that is given in a variable or constant. String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.

### Syntax:

```
[stringprefix]" "
```

### Stringprefix:

- r | R Raw Strings
- u | U Unicode String
- f | F Formatted String

## Raw String:

Python raw string is created by prefixing a string literal with 'r' or 'R'. Python raw string treats backslash (\) as a literal character. This is useful when we want to have a string that contains backslash and don't want it to be treated as an escape character.

It is also handy for regular expressions that make extensive use of backslashes. Example: '\n' is a one-character string with a non-printing newline; r'\n' is a two-character string.

### Syntax:

```
r"String_data"
```

### Example:

#### Program

```
a="Welcome to \nCCIT"
b=r"Welcome to \nCCIT"
print(a)
print(b)
```

#### Output

```
Welcome to
CCIT

Welcome to \nCCIT
```

## Unicode String:

Unicode is the Universal Character Set; each character requires from 1 to 4 bytes of storage. Unicode permits any character in any of the languages in common use around the world.

Unicode is international standard where a mapping of individual characters and a unique number is maintained. As of May 2019, the most recent version of Unicode is 12.1 which contains over 137k characters including different scripts including English, Hindi, Chinese and Japanese, as well as emojis.

### Syntax:

```
u"String_data"
```

### Example:

#### Program

```
msg=u"\u0061 \u2167 \u265E "
print(msg)
```

#### Output

```
a VIII ♟
```

## Formatted String:

Python 3.6 added a new string formatting approach called formatted string literals or “f-strings”. A formatted string literal or f-string is a string literal that is prefixed with 'f' or 'F'.

These strings may contain replacement fields, which are expressions delimited by curly braces {}. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

### Syntax:

```
f"String_data"
```

### Example:

#### Program

```
a="CCIT"
msg=f"Welcome to {a}"
print(msg)
```

#### Output

```
Welcome to CCIT
```

# bytes Objects

The bytes object is one of the core built-in types for manipulating binary data. A bytes object is an immutable sequence of single byte values. Each element in a bytes object is a small integer in the range 0 to 255.

## Syntax:

```
b"bytes_string..."
```

## Bytes method:

The bytes() method returns a immutable bytes object initialized with the given size and data. The bytes() method returns a bytes object which is an immutable (canThe bytes() takes three optional parameters:

- source (Optional) - source to initialize the array of bytes.
- encoding (Optional) - if source is a string, the encoding of the string.
- errors (Optional) - if source is a string, the action to take when the encoding conversion failsnot be modified).

## Syntax:

```
bytes(source,encoding,errors)
```

## Example:

### Program

```
data=b'\xff \x02 \x12'
print(data)

data=bytes(4)
print(data)

data=bytes([12,152,255])
print(data)
```

### Output

```
b'\xff\x02\x12'

b'\x00\x00\x00\x00'

b'\x0c\x98\xff'
```

# bytearray Objects

Python supports another binary sequence type called the bytearray. bytearray objects. There is no dedicated syntax built into Python for defining a bytearray literal, like the 'b' prefix that may be used to define a bytes object. A bytearray object is always created using the bytearray() built-in function

## Bytearray method:

The bytearray() method returns a bytearray object which is an array of the given bytes. The bytearray() method returns a bytearray object which is a mutable (can be modified) sequence of integers. The bytearray() takes three optional parameters:

- source (Optional) - source to initialize the array of bytes.
- encoding (Optional) - if source is a string, the encoding of the string.
- errors (Optional) - if source is a string, the action to take when the encoding conversion fails

## Syntax:

```
bytearray(source,encoding,errors)
```

## Example:

### Program

```
data=bytearray([255,20,11])
print(data)

data[1]=2
print(data)
```

### Output

```
bytearray(b'\xff\x14\x0b')
bytearray(b'\xff\x02\x0b')
```

# Lists

A list is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible. Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets ([]). List is a versatile datatype available in Python. Basically a python list is comma-separated values which are called items. List in python is written within square brackets. Interestingly it's not necessary for items in a list to be of same types. A list is a collection which is ordered and changeable.

Features :

- Lists Are Ordered i.e. its elements can be accessed by using Index.
- Lists Can Contain Arbitrary Objects.
- List Elements Lists Can Be Nested.
- Lists Are Mutable.
- Lists Are Dynamic.

**Syntax:**

```
<list_name>=[value1,value2,value3,...,valueN]
```

**Example:**

Program

```
lst=["Amit","Sumit","Gopal","Harry"]
print(lst)
for n in lst:
    print(n)
```

Output

```
['Amit', 'Sumit', 'Gopal', 'Harry'
']
Amit
Sumit
Gopal
Harry
```

**Example:**

Program

```
lst=[4,3,1,6,2]
s=0
for n in lst:
    s=s+n
print("Sum is ",s)
```

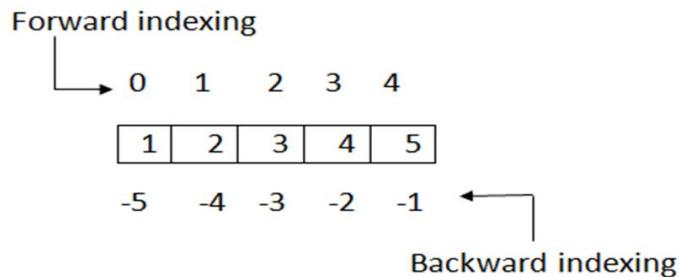
Output

```
Sum is 16
```

## List Indexing:

Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.

List indexing in Python is zero-based: the first item in the list has index 0, the next has index 1, and so on. The index of the last item will be the length of the list minus one.



We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

Trying to access an element other than this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

### Syntax to get a value:

```
<list_name>[index]
```

### Syntax to get a sublist:

```
<list_name>[indexstart: indexstop]
```

### Example:

#### Program

```
Lst = [ 4, 3, 1, 6, 2, 9, 0 ]
print( lst[ 2 ] )
print( lst[ 0 ] )
print( lst[ -1 ] )
print( lst[ 2 : 5 ] )
```

#### Output

```
1
4
0
[1,6,2]
```

## Lists Constructor:

The list() constructor returns a list in Python. The list() constructor takes a single argument an object that could be a sequence (string, tuples) or collection (set, dictionary) or any iterator object.

### Syntax to get a value:

```
<list_name>=list(iterable)
```

### Example:

#### Program

```
vowelString = 'AEIOU'  
lst=list(vowelString)  
print(lst)
```

#### Output

```
[ 'A ', 'E ', 'I ', 'O ', 'U ']
```

## Lists Operator:

Python provides operator that work with lists.

operator	Description
+	operator to combine two lists.
*	repeats a list for the given number of times.
In	test if an item exists in a list or not

### Example:

#### Program

```
lst1=['Amit','Gopal','Sumit']  
lst2=['Harry','Raj']  
  
print(lst1+lst2)  
  
print(lst1*2)  
  
print("Gopal" in lst1)
```

#### Output

```
[ 'Amit', 'Gopal', 'Sumit', 'Harry', 'Raj' ]  
  
[ 'Amit', 'Gopal', 'Sumit', 'Amit', 'Gopal', 'Sumit' ]  
  
True
```

## Lists Functions:

Python provides many functions that are built-in that work with lists.

Function	Description
<b>len(object)</b>	Returns the total number of items in a list
<b>str(object)</b>	Returns the list item as string.
<b>max(object)</b>	returns the largest item in an list.
<b>min(object)</b>	returns the smallest item in an list.

### Example:

#### Program

```
lst=[23,12,45,68,47,12]
print(len(lst))
print(str(lst))
print(max(lst))
print(min(lst))
```

#### Output

```
6
[23, 12, 45, 68, 47, 12]
68
12
```

## Lists method:

Python supplies several built-in methods that can be used to modify lists.

Function	Description
<b>lst.append(obj)</b>	The append() method adds an item to the end of the list.
<b>lst.extend(iterable)</b>	The extend() extends the list by adding all items of a list (passed as an argument) to the end
<b>lst.insert(index, obj)</b>	inserts object <obj> into list a at the specified <index>.
<b>lst.remove(obj)</b>	remove(obj) removes object <obj> from list. If obj isn't in a, an exception is raised
<b>lst.pop(index=-1)</b>	pop() simply removes the last item in the list
<b>lst.index(element)</b>	The index() method searches an element in the list and returns its index.
<b>list.count(element)</b>	The count() method returns the number of occurrences of an element in a list.
<b>lst.copy()</b>	The copy() method returns a shallow copy of the list.
<b>lst.clear()</b>	The clear() method removes all items from the list.
<b>lst.sort()</b>	The sort() method sorts the elements of a given list.
<b>lst.reverse()</b>	The reverse() method reverses the elements of a given list.

**Example:****Program**

```
lst=[23,12,45,12]  
print(lst)
```

```
lst.append(14)  
print(lst)
```

```
lst.extend([23,45])  
print(lst)
```

```
lst.insert(2,35)  
print(lst)
```

```
lst.remove(12)  
print(lst)
```

```
lst.pop()  
print(lst)
```

```
print(lst.index(12))
```

```
print(lst.count(23))
```

```
data=lst.copy()  
print(data)
```

```
lst.sort()  
print(lst)
```

```
lst.reverse()  
print(lst)
```

```
lst.clear()  
print(lst)
```

**Output**

```
[23, 12, 45, 12]
```

```
[23, 12, 45, 12, 14]
```

```
[23, 12, 45, 12, 14, 23, 45]
```

```
[23, 12, 35, 45, 12, 14, 23,  
45]
```

```
[23, 35, 45, 12, 14, 23, 45]
```

```
[23, 35, 45, 12, 14, 23]
```

```
3
```

```
2
```

```
[23, 35, 45, 12, 14, 23]
```

```
[12, 14, 23, 23, 35, 45]
```

```
[45, 35, 23, 23, 14, 12]
```

```
[]
```

## Nested List :

A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth. If required a list can contain items of type list as well as other collection types in it.

### Syntax:

```
<list_name>=[[value1,value2...],[value1,...],valueN]
```

OR

```
<list_name 1>=[value1,value2,valueN]
```

```
<list_name 2>=[value1,value2,valueN]
```

```
<list_name 1>=[list_name 1,list_name 2]
```

### Example:

#### Program

```
nlst=[[1, 3, 5], [2, 4, 6, 3]]  
s=0  
  
for lst in nlst:  
    for n in lst:  
        s=s+n  
  
print("Sum is ",s)
```

#### Output

**Sum is 24**

### Example:

#### Program

```
slst=[["jan-2017","Amit",45000],  
      ["jan-2017","Gopal",15000],  
      ["feb-2017","Amit",25000]]  
  
s=0  
  
for lst in slst:  
    s=s+ lst[2]  
  
print("Total sales is ",s)
```

#### Output

**Total sales is 85000**

# Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed. Tuple is a data structure which is used to store Collection of Items. if the element is itself a mutable datatype like list, its nested items can be changed.

Features :

- Tuple Are Ordered i.e Can Be Accessed by Index
- Tuples Are Immutable.

## Syntax:

```
<tuple_name>=(value1,value2,value3,...,valueN)
```

## Example:

### Program

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

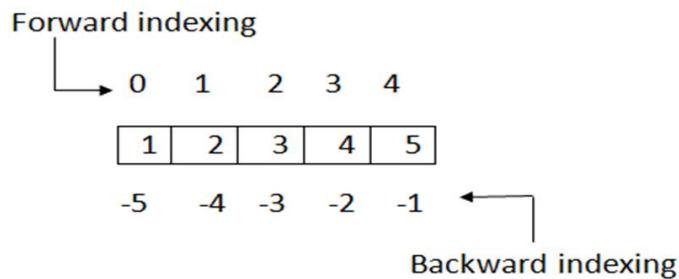
### Output

```
('apple', 'banana', 'cherry')
```

## Tuple Indexing:

Individual elements in a Tuple can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. Tuple indexing is zero-based as it is with strings.

Tuple indexing in Python is zero-based: the first item in the list has index 0, the next has index 1, and so on. The index of the last item will be the length of the tuple minus one.



We can use the index operator [] to access an item in a Tuple. Index starts from 0. So, a Tuple having 5 elements will have index from 0 to 4.

Trying to access an element other than this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

### Syntax to get a value:

```
<tuple_name>[index]
```

### Syntax to get a sublist:

```
<tuple_name>[indexstart: indexstop]
```

### Example:

#### Program

```
tp = ("apple", "banana", "cherry", "mango")
print( tp[ 2 ] )
print( tp[ 0 ] )
print( tp[ -1 ] )
print( tp[ 1 : 3 ] )
```

#### Output

```
cherry
apple
mango
('banana', 'cherry')
```

## Tuple Constructor:

In Python, a tuple is an immutable sequence type. One of the ways of creating tuple is by using the tuple() construct. an iterable (list, range, etc.) or an iterator object.

### Syntax :

```
<tuple_name>=tuple(iterable)
```

### Example:

#### Program

```
vowelString = 'AEIOU'
tp=tuple(vowelString)
print(tp)
```

#### Output

```
('A', 'E', 'I', 'O', 'U')
```

## Tuple Operator:

Python provides operator that work with tuple.

operator	Description
<b>+</b>	operator to combine two tuple.
<b>*</b>	repeats a list for the given number of times.
<b>In</b>	test if an item exists in a list or not

### Example:

#### Program

```
tpl1=('Amit','Gopal','Sumit')
tpl2=('Harry','Raj')

print(tpl1+tpl2)

print(tpl1*2)

print("Gopal" in tpl1)
```

#### Output

```
('Amit', 'Gopal', 'Sumit', 'Harry',
'Raj')

('Amit', 'Gopal', 'Sumit', 'Amit',
'Gopal', 'Sumit')

True
```

## Tuple Functions:

Python provides many functions that are built-in that work with Tuple.

Function	Description
<b>len(object)</b>	Returns the total number of items in a tuple
<b>str(object)</b>	Returns the tuple item as string.
<b>max(object)</b>	returns the largest item in an tuple.
<b>min(object)</b>	returns the smallest item in an tuple.

**Example:**

Program

```
tpl=(23,12,45,68,47,12)
print(len(tpl))
print(str(tpl))
print(max(tpl))
print(min(tpl))
```

Output

```
6
(23, 12, 45, 68, 47, 12)
68
12
```

**Tuple method:**

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Function	Description
<b>tpl.count(x)</b>	Returns the number of items x
<b>tpl.index(x)</b>	Returns the index of the first item that is equal to x

**Example:**

Program

```
tpl=(23,12,45,68,47,12)
print(tpl.count(12))
print(tpl.index(12))
```

Output

```
2
1
```

# Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

A Python set is similar to mathematical Set. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table. There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Features :

- Sets are unordered.
- Set elements are unique. Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.

## Syntax:

```
<set_name>={value1,value2,value3,...,valueN}
```

## Example:

### Program

```
thissets = {"apple", "banana", "cherry"}  
print(thissets)
```

### Output

```
{'apple', 'banana', 'cherry'}
```

## set Constructor:

The set() builtin creates a Python set from the given iterable. set() takes a single optional parameter iterable (optional) a sequence (string, tuple, etc.) or collection (set, dictionary, etc.) or an iterator object to be converted into a set. set() returns an empty set if no parameters are passed and a set constructed from the given iterable parameter.

## Syntax:

```
<set_name>=set(iterable)
```

**Example:****Program**

```
lst=["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]
Days=set(lst)
print(Days)
```

**Output**

```
{"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"}
```

**Set Operator:**

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

<b>operator</b>	<b>Description</b>
	Union of A and B is a set of all elements from both sets.
&	Intersection of A and B is a set of elements that are common in both sets.
-	Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of element in B but not in A.
^	Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both.

**Example:****Program**

```
daysA={"Mon", "Tue", "Wed", "Sat", "Sun"}
daysB={"Thu", "Fri", "Sat", "Sun"}

print(daysA | daysB)

print(daysA & daysB)

print(daysA - daysB)

print(daysA ^ daysB)
```

**Output**

```
{'Fri', 'Sat', 'Thu', 'Sun', 'Wed', 'Mon', 'Tue'}
{'Sun', 'Sat'}
{'Mon', 'Wed', 'Tue'}
{'Fri', 'Thu', 'Wed', 'Mon', 'Tue'}
```

## set Functions:

Python provides many functions that are built-in that work with set.

Function	Description
<b>len(object)</b>	Returns the total number of items in a set
<b>str(object)</b>	Returns the set item as string.
<b>max(object)</b>	returns the largest item in an set.
<b>min(object)</b>	returns the smallest item in an set.
<b>sum(object)</b>	The sum() function adds the items of an iterable and returns the sum.
<b>sorted(iterable, reverse=False)</b>	The sorted() function returns a sorted list from the items in an iterable. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator. reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

### Example:

#### Program

```
num={23,12,45,56,35,31,21}

print(len(num))

print(str(num))

print(max(num))

print(min(num))

print(sum(num))

print(sorted(num))

print(sorted(num,reverse=True))
```

#### Output

```
7
{35, 12, 45, 21, 23, 56, 31}
56
12
223
[12, 21, 23, 31, 35, 45, 56]
[56, 45, 35, 31, 23, 21, 12]
```

## Set method:

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with set objects.

Function	Description
<b>set.add(x)</b>	The set add() method adds a given element to a set. If the element is already present, it doesn't add any element.
<b>set.remove(x)</b>	The remove() method searches for the given element in the set and removes it.
<b>set.discard(x)</b>	The discard() method removes a specified element from the set (if present).
<b>set.update(iterable)</b>	The Python set update() method updates the set, adding items from other iterables.
<b>set.pop()</b>	The pop() method removes an arbitrary element from the set and returns the element removed.
<b>set.copy()</b>	The copy() method returns a shallow copy of the set.
<b>set.clear()</b>	The clear() method removes all elements from the set.
<b>set.difference(setB)</b>	The difference() method returns the set difference of two sets.
<b>set.intersection(setB)</b>	The intersection() method returns a new set with elements that are common to all sets.
<b>set.symmetric_difference(setB)</b>	The Python symmetric_difference() method returns the symmetric difference of two sets.
<b>set.union(setB)</b>	The Python set union() method returns a new set with distinct elements from all the sets.
<b>set.isdisjoint(setB)</b>	The isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False
<b>set.issubset(setB)</b>	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.

**Example:****Program**

```

numA={23,12,45,56,35,31,21}
numB={23,12,89,78,35,87}

numA.add(26)
print(numA)

numA.remove(26)
print(numA)

numA.update([26,32])
print(numA)

numA.discard(26)
print(numA)

numA.pop()
print(numA)

print(numA.difference(numB))

print(numA.intersection(numB))

print(numA.symmetric_difference(numB))

print(numA.union(numB))

print(numA.isdisjoint(numB))

print(numA.issubset(numB))

numC=numA.copy()
print(numC)

numA.clear()
print(numA)

```

**Output**

```

{35, 12, 45, 21, 23, 56, 26, 31}

{35, 12, 45, 21, 23, 56, 31}

{32, 35, 12, 45, 21, 23, 56, 26, 31}

{32, 35, 12, 45, 21, 23, 56, 31}

{35, 12, 45, 21, 23, 56, 31}

{56, 21, 45, 31}

{35, 12, 23}

{45, 78, 21, 87, 56, 89, 31}

{35, 12, 45, 78, 21, 87, 23, 56, 89, 31}

False

False

{35, 21, 23, 56, 12, 45, 31}

set()

```

# Frozen sets

Python provides another built-in type called a frozenset, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset. Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary. Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of frozen set remains the same after creation.

Frozensets can be created using the function `frozenset()`. The `frozenset()` method returns an immutable frozenset object initialized with elements from the given iterable. The `frozenset()` method optionally takes a single parameter

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

## Syntax:

```
<set_name>=frozenset(iterable)
```

## Example:

### Program

```
days=frozenset(["Mon", "Tue", "Wed", "Thu", "Fri",
                 "Sat", "Sun"])
print(days)
```

### Output

```
frozenset({'Wed', 'Fri',
            'Mon', 'Tue', 'Sun', 'Sat',
            'Thu'})
```

## Frozensets Functions:

Python provides many functions that are built-in that work with Frozenset.

Function	Description
<b>len(object)</b>	Returns the total number of items in a frozen set
<b>str(object)</b>	Returns the frozen set item as string.
<b>max(object)</b>	returns the largest item in an frozen set.
<b>min(object)</b>	returns the smallest item in an frozen set.
<b>sum(object)</b>	The sum() function adds the items of an iterable and returns the sum.
<b>sorted(iterable, reverse=False)</b>	The sorted() function returns a sorted list from the items in an iterable. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

### Example:

#### Program

```
num= frozenset([12,5,35,1,4,14])

print(len(num))

print(str(num))

print(max(num))

print(min(num))

print(sum(num))

print(sorted(num))

print(sorted(num,reverse=True))
```

#### Output

```
6
frozenset({1, 35, 4, 5, 12, 14})
35
1
71
[1, 4, 5, 12, 14, 35]
[35, 14, 12, 5, 4, 1]
```

## Frozenset method:

There are many frozenset methods, some of which we have already used above. Here is a list of all the methods that are available with frozenset objects.

Function	Description
<b>set.copy()</b>	The copy() method returns a shallow copy of the frozenset.
<b>set.difference(setB)</b>	The difference() method returns the set difference of two sets.
<b>set.intersection(setB)</b>	The intersection() method returns a new set with elements that are common to all frozensets.
<b>set.symmetric_difference(setB)</b>	The Python symmetric_difference() method returns the symmetric difference of two frozensets.
<b>set.union(setB)</b>	The Python set union() method returns a new set with distinct elements from all the frozensets.
<b>set.isdisjoint(setB)</b>	The isdisjoint() method returns True if two frozensets are disjoint frozensets. If not, it returns False
<b>set.issubset(setB)</b>	The issubset() method returns True if all elements of a frozensets are present in another frozensets (passed as an argument). If not, it returns False.

### Example:

#### Program

```

numA=frozenset({23,12,45,56,35,31,21})

numB=frozenset({23,12,89,78,35,87})

print(numA.difference(numB))

print(numA.intersection(numB))

print(numA.symmetric_difference(numB))

print(numA.union(numB))

print(numA.isdisjoint(numB))

print(numA.issubset(numB))

numC=numA.copy()

print(numC)

```

#### Output

```

frozenset({56, 45, 21, 31})

frozenset({35, 12, 23})

frozenset({45, 78, 21, 87, 56, 89, 31})

frozenset({35, 12, 45, 78, 21, 87, 23, 56, 89, 31})

False

False

frozenset({35, 21, 23, 56, 12, 45, 31})

```

# Dictionary

Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value. Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair. Dictionaries are optimized to retrieve values when the key is known.

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma. An item has a key and the corresponding value expressed as a pair, key: value. While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Features :

- It is mutable. i.e. its elements can be changed.
- It is Dynamic. i.e. elements can be added / removed at runtime.
- It is Ordered . i.e. Its elements can be accessed by using a key.
- It can be nested i.e. A dictionary can contain another collection.

## Syntax:

```
<dictionary_name>={key1:value1,key2:value2,.....,keyN:valueN}
```

- The key must be unique.
- Value is accessed by key.
- Value can be updated while key cannot be changed.

## Example:

### Program

```
data={'first_name':'Amit','last_name':'Jain',
'age':24}
print(data)
```

### Output

```
{'first_name': 'Amit',
'last_name': 'Jain', 'age':
24}
```

## Accessing Dictionary Values:

Dictionary values can be accessed by using keys .A value is retrieved from a dictionary by specifying its corresponding key in square brackets ([ ]). While indexing is used with other container types to access values, dictionary uses keys.

### Syntax:

```
<dictionary_name>[key]
```

### Example:

#### Program

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
print(my_dict["name"])

print(my_dict[1])
```

#### Output

```
John
[2, 4, 3]
```

## Dictionary Constructor:

The dict( ) constructor creates a dictionary in Python. A keyword argument is an argument preceded by an identifier (eg. name=). Hence, the keyword argument of the form kwarg=value is passed to the dict( ) constructor to create dictionaries. The dict( ) doesn't return any value (returns None).

### Syntax:

```
<dict_name>=dict(**kwarg)
```

### Example:

#### Program

```
data=dict(first_name="Amit",last_name="Jain",
age=31)
print(data)
```

#### Output

```
{"Mon", "Tue", "Wed", "Thu", "Fri",
,"Sat", "Sun"}
```

## Dictionary Functions:

Python provides many functions that are built-in that work with Dictionary.

Function	Description
<b>len(object)</b>	Returns the total number of items in a dictionary
<b>str(object)</b>	Returns the dictionary item as string.
<b>max(object)</b>	returns the largest key in an dictionary.
<b>min(object)</b>	returns the smallest key in an dictionary.
<b>sorted(iterable, reverse=False)</b>	The sorted() function returns a sorted list from the items in an iterable. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator. reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

### Example:

#### Program

```
data={2010:"Amit",3021:"Sumit",4215:"Gopal",2531:"Harry"

print(len(data))

print(max(data))

print(min(data))

print(sum(data))

print(sorted(data))

print(sorted(data,reverse=True))
```

#### Output

```
4
4215
2010
11777
[2010, 2531, 3021, 4215]
[4215, 3021, 2531, 2010]
```

## Dictionary method:

Methods that are available with dictionary are tabulated below.

Function	Description
<b>dict.get(key)</b>	The get() method returns the value for the specified key if key is in dictionary.
<b>dict.keys()</b>	The keys() method returns a view object that displays a list of all the keys in the dictionary.
<b>dict.items()</b>	The items() method returns a view object that displays a list of dictionary's (key, value) tuple pairs.
<b>dict.popitem()</b>	The popitem() returns and removes an arbitrary element (key, value) pair from the dictionary.
<b>dict.setdefault(key, default_value)</b>	The setdefault() method returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.
<b>dict.pop(key, default_value)</b>	The pop() method removes and returns an element from a dictionary having the given key.
<b>dict.values()</b>	The values() method returns a view object that displays a list of all the values in the dictionary.
<b>dict.update(dict)</b>	The update() method updates the dictionary with the elements from the another dictionary object or from an iterable of key/value pairs.
<b>dict.copy()</b>	They copy() method returns a shallow copy of the dictionary.
<b>dict.clear()</b>	The clear() method removes all items from the dictionary.
<b>dict.fromkeys(sequence, value)</b>	The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.

**Example:****Program**

```

data={2010:"Amit",3021:"Sumit",4215:"Gopal",2
531:"Harry"}
print(data.get(2010))

print(data.keys())

print(data.items())

print(data.popitem())
print(data)

print(data.setdefault(2531,"Harry"))
print(data)

print(data.pop(2532,None))

print(data.values())

data.update({4251:"Raj"})
print(data)

data2=data.copy()
print(data2)

data.clear()
print(data)

```

**Output**

```

Amit

dict_keys([2010, 3021, 4215,
2531])

dict_items([(2010, 'Amit'),
(3021, 'Sumit'), (4215,
'Gopal'), (2531, 'Harry')])

(2531, 'Harry')
{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal'}

Harry
{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal', 2531: 'Harry'}

None

dict_values(['Amit', 'Sumit',
'Gopal', 'Harry'])

{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal', 2531: 'Harry',
4251: 'Raj'}

{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal', 2531: 'Harry',
4251: 'Raj'}

{}

```

## Processing Dictionary:

Items of dictionary can be processed by using for loop. If for loop is applied for dictionary object it return key of each item.

### Example:

#### Program

```
dit = {'Name': 'Amit Jain', 'Age':  
    7, 'City': 'Amravati'};  
  
for key in dit:  
    print(key);
```

#### Output

```
Name  
Age  
City
```

### Example:

#### Program

```
dit = {'Name': 'Amit Jain', 'Age':  
    7, 'City': 'Amravati'};  
  
for key, val in dic.items():  
    print(key, " : ", val);
```

#### Output

```
Name : Amit Jain  
Age : 7  
City : Amravati
```

# Map Function

The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results. It applies a given function to each item of an iterable (i.e Collection for ex: list, tuple etc.).The map() function applies a given to function to each item of an iterable and returns a list of the results. The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) and so on.

Note: You can pass more than one iterable to the map() function.

## Syntax:

```
map(function, iterable, ...)
```

## Parameter:

- function - map() passes each item of the iterable to this function.
- Iterable - iterable which is to be mapped.

## Example:

### Program

```
def myfunc(nm):
    return nm.upper()

names=['amit', 'raj', 'mohan' ]
itr = map( myfunc, names )
lst=list(itr)
print(lst)
```

### Output

```
[ 'AMIT', 'RAJ', 'MOHAN' ]
```

**Example:****Program**

```
def myfunc(a, b):
    return a + " " + b

itr = map(myfunc, ('amit', 'gopal', 'raj',
                   'mona'), ('jain', 'pandey', 'joshi'))
lst=list(itr)
print(lst)
```

**Output**

```
['amit jain', 'gopal pandey',
 'raj mona joshi']
```

**Example:****Program**

```
print("Enter nos.")
nos=input().split()
print(nos)

itr=map(int,nos)
lst=list(itr)
print(lst)
```

**Output**

```
Enter nos.
5 10 20 10 5 2
['5', '10', '20', '10', '5', '2']
[5, 10, 20, 10, 5, 2]
```

**Example:****Program**

```
fnames=('amit', 'gopal', 'raj')
lnames=('joshi', 'patil', 'pandey')

itr = map(lambda fn,ln : fn+" "+ln , fnames ,
          lnames)
lst=list(itr)
print(lst)
```

**Output**

```
['amit joshi', 'gopal patil',
 'raj pandey']
```

# Array

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character.

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

## Syntax:

```
array(typecode, initializer)
```

A new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, `TypeError` is raised.

## Accessing Array Values:

Accessing elements is the same as accessing Strings in Python. You pass the index values and hence can obtain the values as needed.

### Syntax:

```
<array_name>[index]
```

### Example:

#### Program

```
import array
arr=array.array('i',[1,2,3,4,5,6,7])
print(arr[5])
print(arr[2:7])
print(arr[:3])
```

#### Output

```
6
array('i', [3, 4, 5, 6, 7])
array('i', [1, 2, 3])
```

## Array method:

Methods that are available with array are tabulated below.

Function	Description
<b>array.append(x)</b>	Append a new item with value x to the end of the array.
<b>array.count(x)</b>	Return the number of occurrences of x in the array.
<b>array.extend(iterable)</b>	Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, TypeError will be raised.
<b>array.index(x)</b>	Return the smallest i such that i is the index of the first occurrence of x in the array.
<b>array.insert(i, x)</b>	Insert a new item with value x in the array before position i. Negative values are treated as being relative to the end of the array.
<b>array.pop([i])</b>	Removes the item with the index i from the array and returns it.
<b>array.remove(x)</b>	Remove the first occurrence of x from the array.
<b>array.reverse()</b>	Reverse the order of the items in the array.

**array.tolist()**

Convert the array to an ordinary list with the same items.

**Example:****Program**

```
import array

arr=array.array('i',[1,2,3,4,5,6])
print(arr)
arr.append(3)
print(arr)
print(arr.count(3))
arr.extend([2,5])
print(arr)
print(arr.index(2))
arr.insert(2,5)
print(arr)
print(arr.pop())
arr.remove(3)
print(arr)
arr.reverse()
print(arr)
print(arr.tolist())
```

**Output**

```
array('i', [1, 2, 3, 4, 5, 6])
array('i', [1, 2, 3, 4, 5, 6, 3])
2
array('i', [1, 2, 3, 4, 5, 6, 3, 2, 5])

1
array('i', [1, 2, 5, 3, 4, 5, 6, 3, 2, 5])
5
array('i', [1, 2, 5, 4, 5, 6, 3, 2])
array('i', [2, 3, 6, 5, 4, 5, 2, 1])
[2, 3, 6, 5, 4, 5, 2, 1]
```

# Modules

---

---

# Modules

A module is simply a Python file, where classes, functions and variables are defined. These modules can be imported in our project whenever required. Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: example.py, is called a module and its module name would be example. We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code. We can define our most used functions in a module and import it, instead of copying their definitions into different programs. Let us create a module. Type the following and save it as example.py.

There are 2 types of modules:

- Built-in Modules
- User-defined Modules

## Built-in Modules

Some popular modules in Python are.

1. math : for mathematical functions
2. threading : for multithreaded applications.
3. collections : for additional collections classes.
4. os : operating system related functions
5. re : for text processing
6. random : for random number generation
7. pickle : is used for serializing and de-serializing Python objects.
8. nltk : natural language processing.
9. datetime : is used to perform operations on date and time data.
10. tkinter : for GUI interface..
11. sockets : for network programming.
12. Etc

## Import Statement

It is used to import a module. We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this. We can import a module using import statement and access the definitions inside it using the dot operator.

### Syntax:

```
import <module_name>
```

### Example:

#### Program

```
import math
print("Pi value: ",math.pi)
```

#### Output

```
Pi value 3.141592653589793
```

## from import Statement

It is used to import particular attribute from a module. We can import specific names from a module without importing the module as a whole. We can import all names (definitions) from a module

### Syntax:

```
from <modulename> import <attributename>,..
from <modulename> import *
```

### Example:

#### Program

```
from math import pi
print("Pi value: ",pi)
```

#### Output

```
Pi value 3.141592653589793
```

## Renaming a module:

We can create an alias when we import a module. This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names

### Syntax:

```
import <modulename> as <aliasname>
from <module_name> import <name> as <alt_name>
```

### Example:

Program

```
import math as m
print("Pi value: ",m.pi)
```

Output

```
Pi value 3.141592653589793
```

## User-defined Modules

It is module that is created by the user so that module can be used by himself or by other users in other programs.

Step to create a module

- Create a file having same name as modulename.
- In that file we can define any type of components such as functions , variables, classes , objects etc.

**Example:****Program**

```

mylib.py
-----
class Rectangle:

    def setdimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)

class Circle:

    def setradius(this,n):
        this.r=n

    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)

def display(n):
    for i in range(1,n+1):
        print(i)

demo.py
-----
import mylib
a=mylib.Circle()
b=mylib.Rectangle()
a.setradius(5)
b.setdimension(5,10)
a.area()
b.area()
mylib.display(5)

```

**Output**

```

Area is  78.5
Area is  50
1
2
3
4
5

```

Note : If import statement is used to import a module then components of that module can be accessed by using syntax: **moduleName.ComponentName**

**Example:****Program**

```
mylib.py
-----
class Rectangle:
    def setdimension(self,x,y):
        self.length=x
        self.breadth=y
    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)
class Circle:
    def setradius(this,n):
        this.r=n
    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)
def display(n):
    for i in range(1,n+1):
        print(i)
```

**demo.py**

```
from mylib import Circle
a=Circle()
a.setradius(5)
a.area()
```

**Output**

Area is 78.5

**Note :** If from import statement is used to import an Component of a module then components of that module can be accessed by using **syntax: ComponentName**

We can use \* to import all components from a module.

**Syntax: from <module> import \***

**Example:****Program**

```
mylib.py
-----
class Rectangle:

    def setdimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)

class Circle:

    def setradius(this,n):
        this.r=n

    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)

def display(n):
    for i in range(1,n+1):
        print(i)
```

**demo.py**

```
import mylib as lib

a=lib.Circle()
a.setradius(5)
a.area()
```

**Output**

Area is 78.5

**Note :** If modules or components have lengthy names then we can use some short names for them  
**syntax:**

**AliasModuleName.ComponentName**  
**AliasCompName**

## Python Module Search Path

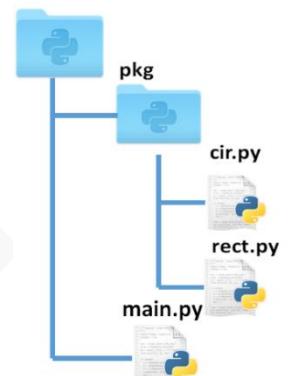
While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) into a list of directories defined in sys.path. The search is in this order.

- The current directory.
- PYTHONPATH (an environment variable with a list of directory).
- The installation-dependent default directory.

## Python Packages

A package is a collection of Python modules. Packages allow for a hierarchical structuring of the module namespace using dot notation. In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names.

Creating a package is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure.



### Example:

#### Program

```

cir.py
-----
class Circle:
    def setradius(this,n):
        this.r=n
    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)
rect.py
-----
class Rectangle:
    def setdimension(self,x,y):
        self.length=x
        self.breadth=y
    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)
  
```

```

main.py
-----
from pkg import Circle,Rectangle
a=cir.Circle()
a.setradius(5)
a.area()
b=rect.Rectangle()
b.setdimension(5,2)
b.area()
  
```

#### Output

```

Area is  78.5
Area is  10
  
```

## Packages \_\_init\_\_.py

A package is a directory of Python modules containing an additional `__init__.py` file. The `__init__.py` file indicate that it is a python package directory. If a file named `__init__.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

### Example:

#### Program

```

cir.py
-----
class Circle:
    def setradius(this,n):
        this.r=n

    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)

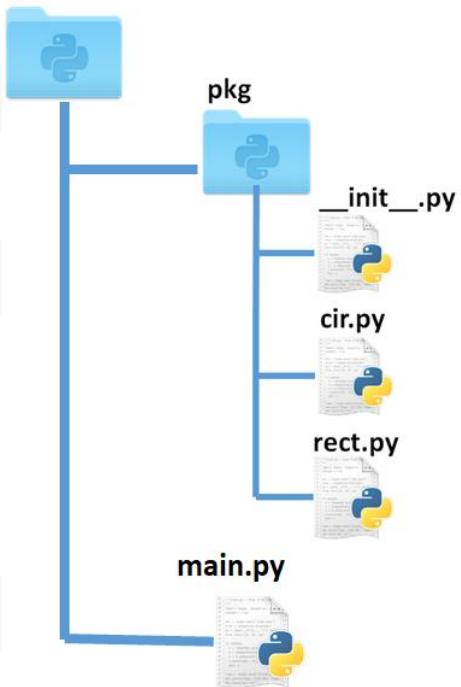
rect.py
-----
class Rectangle:
    def setdimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)

__init__.py
-----
import pkg.Circle,pkg.Rectangle

file=["Circle","Rectangle"]

```



```

main.py
-----
import pkg
print(pkg.file)

a=pkg.cir.circle()
a.setradius(5)
a.area()

b=pkg.rect.Rectangle()
b.setdimension(5,2)
b.area()

```

### Output

```

["Circle","Rectangle"]
Area is  78.5
Area is  10

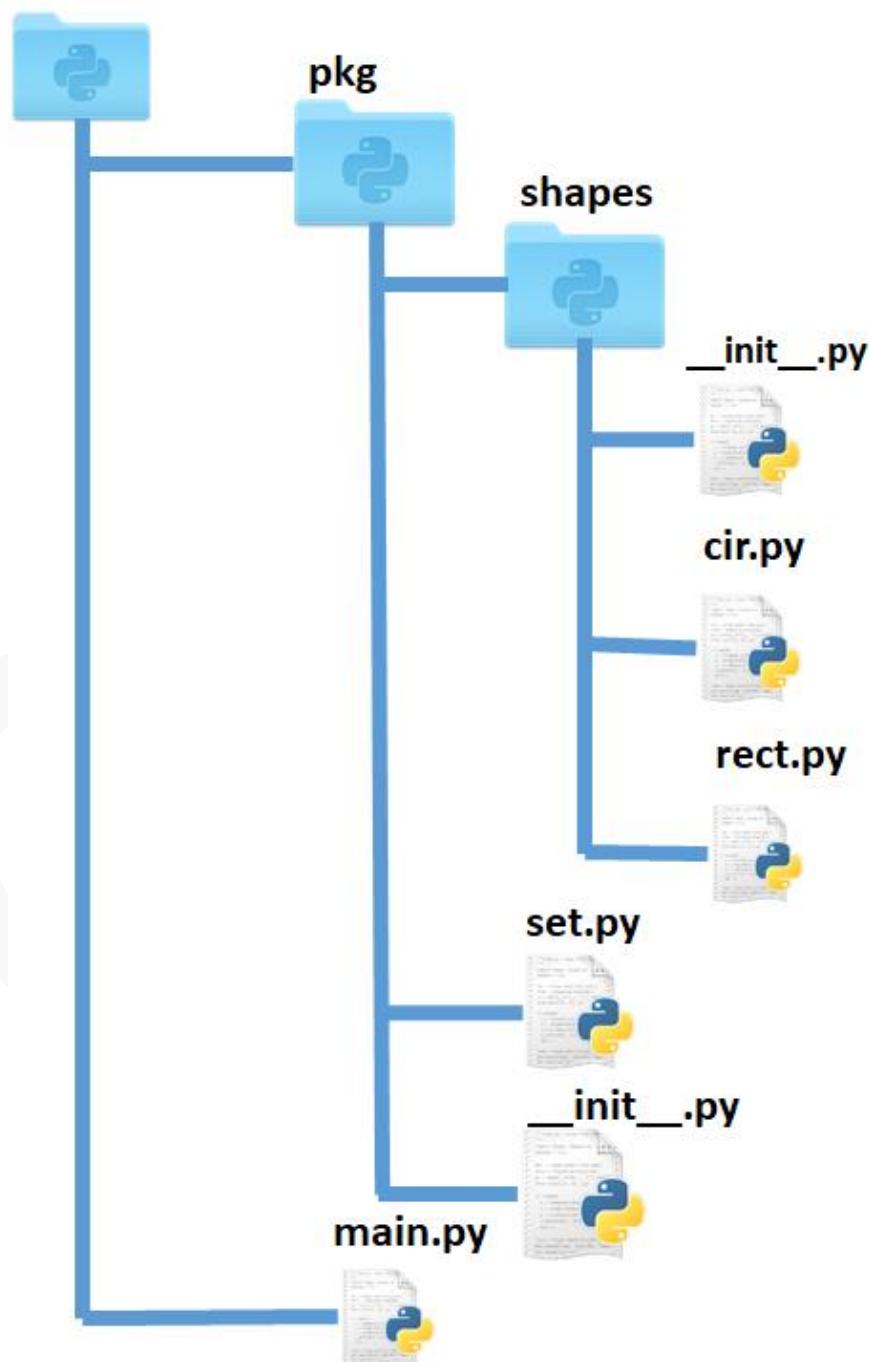
```

## Packages Subpackages

Packages can contain nested subpackages to arbitrary depth. When you import a package, only variables / functions/ classes of that package are directly visible, not sub-packages .

### Example:

#### Program



## Program

```
pkg/shapes/cir.py
```

```
class Circle:
    def setradius(this,n):
        this.r=n
    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)
```

```
pkg/shapes/rect.py
```

```
class Rectangle:
    def setdimension(self,x,y):
        self.length=x
        self.breadth=y
    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)
```

```
pkg/shapes/__init__.py
```

```
import pkg.shapes.cir
import pkg.shapes.rect
```

```
pkg/__init__.py
```

```
import pkg.set
```

```
pkg/set.py
```

```
class Set:
    def __init__(self,a,b,c):
        self.N1=a
        self.N2=b
        self.N3=c
    def sum(self):
        s=self.N1+self.N2+self.N3
        print("Sum is ",s)
```

```
main.py
```

```
from pkg import *
from pkg.shapes import *
b=set.Set(2,3,5)
b.sum()
c=cir.Circle(5)
c.area()
a=rect.Rectangle(5,2)
a.area()
```

## Output

```
Sum is 10
Area is 78.5
Area is 10
```

# Main Functions in Python

Many programming languages have a special function that is automatically executed when an operating system starts to run a program. This function is usually called main() and must have a specific return type and arguments according to the language standard. On the other hand, the Python interpreter executes scripts starting at the top of the file, and there is no specific function that Python automatically executes.

Nevertheless, having a defined starting point for the execution of a program is useful for understanding how a program works. Python programmers have come up with several conventions to define this starting point.

Python offers other conventions to define the execution point. One of them is using the main() function and the \_\_name\_\_ property of a python file.

## \_\_name\_\_

The \_\_name\_\_ variable is a special builtin Python variable that shows the name of the current module. It has different values depending on where we execute the Python file. If the python interpreter is running that module (the source file) as the main program, it sets the special \_\_name\_\_ variable to have a value "\_\_main\_\_". If this file is being imported from another module, \_\_name\_\_ will be set to the module's name. Module's name is available as value to \_\_name\_\_ global variable.

### Example:

#### Program

```
print(f"Module Name : {__name__}")
```

#### Output

```
Module Name : __main__
```

When we run the program as a script, the value of the variable \_\_name\_\_ is set to \_\_main\_\_. So the output

### Example:

#### Program

```
first_module.py
-----
print(f"Module Name : {__name__}")

second_module.py
-----
import first_module
print(f"Module Name : {__name__}")
```

#### Output

```
Module Name : first_module
Module Name : __main__
```

In above program we run second\_module.py file. In that file we imported first\_module. So that this program the interpreter well first execute all statements in first\_module and than it well execute second\_module.

## Using if conditional with `__name__`

Now that we have understood how `__name__` variable is assigned values, we can use the if conditional clause to run the same Python file differently in different contexts.

### Example:

#### Program

```
first_module.py
-----
def func():
    print(f"Module Name : {__name__}")

if __name__ == "__main__":
    func()

second_module.py
-----
import first_module
print(f"Module Name : {__name__}")
```

#### Output

```
Module Name : __main__
```

When second\_modue is run, the if statement condition in first-module is false. So function func from first\_module will not be executed. Only print statement of second\_module will be executed.

### Example:

#### Program

```
first_module.py
-----
def func():
    print(f"Module Name : {__name__}")

if __name__ == "__main__":
    func()
```

#### Output

```
Module Name : __main__
```

When program is run, the if statement condition is true. So function func will be executed.

# Exception Handling

# Exception Handling

It is a technique of handling runtime errors in structured way. An exception is an error that happens during execution of a program. When an error occurs, Python generates an exception representing that error that can be handled, which avoids your program from being crashed. In case if exception is not handled, then the code is not executed further and hence execution stops.

Python has many built-in exceptions which forces your program to output an error when something goes wrong. When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash. Python (interpreter) raises exceptions when it encounters errors. For example: divided by zero.

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (`FileNotFoundException`), dividing a number by zero (`ZeroDivisionError`), module we try to import is not found (`ImportError`) etc.

Whenever these type of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

## Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur.

Exception	Cause of Error
<b>AssertionError</b>	Raised when assert statement fails.
<b>AttributeError</b>	Raised when attribute assignment or reference fails.
<b>EOFError</b>	Raised when the input() function hits end-of-file condition.
<b>FloatingPointError</b>	Raised when a floating point operation fails.
<b>GeneratorExit</b>	Raised when a generator's close() method is called.
<b>ImportError</b>	Raised when the imported module is not found.
<b>IndexError</b>	Raised when index of a sequence is out of range.
<b>KeyError</b>	Raised when a key is not found in a dictionary.

<b>KeyboardInterrupt</b>	Raised when the user hits interrupt key (Ctrl+c or delete).
<b>MemoryError</b>	Raised when an operation runs out of memory.
<b>NameError</b>	Raised when a variable is not found in local or global scope.
<b>NotImplementedError</b>	Raised by abstract methods.
<b>OSError</b>	Raised when system operation causes system related error.
<b>OverflowError</b>	Raised when result of an arithmetic operation is too large to be represented.
<b>ReferenceError</b>	Raised when a weak reference proxy is used to access a garbage collected referent.
<b>RuntimeError</b>	Raised when an error does not fall under any other category.
<b>StopIteration</b>	Raised by next() function to indicate that there is no further item to be returned by iterator.
<b>SyntaxError</b>	Raised by parser when syntax error is encountered.
<b>IndentationError</b>	Raised when there is incorrect indentation.
<b>TabError</b>	Raised when indentation consists of inconsistent tabs and spaces.
<b>SystemError</b>	Raised when interpreter detects internal error.
<b>SystemExit</b>	Raised by sys.exit() function.
<b>TypeError</b>	Raised when a function or operation is applied to an object of incorrect type.
<b>UnboundLocalError</b>	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
<b>UnicodeError</b>	Raised when a Unicode-related encoding or decoding error occurs.
<b>ValueError</b>	Raised when a function gets argument of correct type but improper value.
<b>ZeroDivisionError</b>	Raised when second operand of division or modulo operation is zero.

## try and except

The code where run time error may occur must be enclosed in the try block. The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause.

Except statement specifies the exception which occurred. In case that exception is occurred, the corresponding statement will be executed. There can be multiple except statement with a single try block.

**Syntax:**

```
try:  
    Statement  
    -----  
except exceptionX :  
    statement  
    -----  
except exceptionY :  
    statement  
    -----
```

**Example:**

Program

```
try:  
    print("Start")  
    z=4/0  
    print(z)  
except ZeroDivisionError:  
    print("divide by zero")  
except ValueError:  
    print("Value Error")
```

Output

```
Start  
divide by zero
```

## try and except...else block

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions. This code which will be executed in the scenario if no exception occurs in the try block.

### Syntax:

```
try:
    Statement
    -----
    except exceptionX :
        statement
        -----
    else :
        statement
        -----
```

### Example:

#### Program

```
try:
    print("Start")
    z=4/2
    print(z)

except ZeroDivisionError:
    print("divide by zero")

except ValueError:
    print("Value Error")

else:
    print("No error")
```

#### Output

```
Start
2.0
No error
```

## try and except...finally block

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources. finally enables you to execute sections of code that should always run, with or without any previously encountered exceptions. This code which be executed, whether exception occurs or not.

### Syntax:

```
try:
    Statement
    -----
    except exceptionX :
        statement
        -----
    finally:
        statement
        -----
```

### Example:

#### Program

```
try:
    print("Start")
    z=4/2
    print(z)

except ZeroDivisionError:
    print("divide by zero")

except ValueError:
    print("Value Error")

else:
    print("No error")
```

#### Output

```
Start
2.0
No error
```

## raise statement

The raise statement allows the programmer to force a specified exception to occur. An exception can be raised by using the raise statement.

### Syntax:

```
raise ExceptionObject
```

### Example:

#### Program

```
try:
    age = int(input("Enter the age?"))
    if age<0:
        raise ValueError;
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

#### Output

```
Start
2.0
No error
```

### Example:

#### Program

```
try:
    r = int(input("Enter Radius"))
    if r<0:
        raise ValueError;
    else:
        a=3.14*r**2
        print("Area is ",a)

except ValueError:
    print("Radius cannot be Negative")
```

#### Output

```
Enter Radius5
Area is 78.5
```

## Custom Exception

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class. Most of the built-in exceptions are also derived from this class. We can create our exceptions that can be raised from the program and caught using the `except` clause. The custom exception class must be created from base class `Exception`.

### Example:

#### Program

```
class AgeError(Exception):
    pass

try:
    age = int(input("Enter the age?"))
    if age<0:
        raise AgeError()
    else:
        print("the age is valid")
except AgeError:
    print("The age is not valid")
```

#### Output

```
Enter the age? -2
The age is not valid
```

# Iterators

---

---

# Iterators

It is an object which will return data, one element at a time from an iterable object. An object is called iterable if we can get an iterator from it. 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. Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight. When we reach the end and there is no more data to be returned, it will raise StopIteration.

Technically speaking, Python iterator object must implement two special methods, `__iter__()` and `__next__()`, collectively called the iterator protocol. An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.

## iter function

The `iter()` function creates an object which can be iterated one element at a time. These objects are useful when coupled with loops like for loop, while loop.

Parameters

- object - object whose iterator has to be created (can be sets, tuples, etc.)
- sentinel (optional) - special value that is used to represent the end of a sequence

### Syntax:

```
iter(object, sentinel)
```

## next function

The `next()` function returns the next item from the iterator.

Parameters

- iterator - `next()` retrieves next item from the iterator
- default (optional) - this value is returned if the iterator is exhausted (there is no next item)

### Syntax:

```
next(iterator, default)
```

**Example:****Program**

```
lst=[1,6,2,16,7,22]
p=iter(lst)
print(next(p))
print(next(p))
print(next(p))
print(next(p))
print(next(p))
print(next(p))
print(next(p))
print(next(p))
```

**Output**

```
1
6
2
16
7
22
StopIteration Exception
```

**Example:****Program**

```
tp=("Amit","Sumit","Gopal","Raja")
t=iter(tp)

for i in t:
    print(i)
```

**Output**

```
Amit
Sumit
Gopal
Raja
```

## Creating an Iterable class

To create an object/class as an iterator you have to implement the methods `_iter_()` and `_next_()` to your object.

### `_iter_()`

- This method is similar to constructor .
- It is used for initialization .
- but must always return the iterable object itself.

### `_next_()`

- This method allows us to do operations
- It must return the next item in the sequence.

### Example:

#### Program

```
class myctr:

    def __iter__(self):
        self.ctr=0
        return self

    def __next__(self):
        self.ctr=self.ctr+1
        return self.ctr


a=myctr()
lst=iter(a)
print(next(lst))
print(next(lst))
print(next(lst))
print(next(lst))
```

#### Output

```
1
2
3
4
```

## StopIteration Exception

To prevent the iteration to go on forever, we can use the raise StopIteration statement. In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times

### Example:

#### Program

```
class MyNumbers:

    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

obj = MyNumbers()
myiter = iter(obj)

for x in myiter:
    print(x)
```

#### Output

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

**Example:****Program**

```
class myctr:
    def __init__(self,n):
        self.n=n

    def __iter__(self):
        self.ctr=0
        return self

    def __next__(self):
        if self.ctr==self.n:
            raise StopIteration
        self.ctr=self.ctr+1
        return self.ctr

for i in myctr(10):
    print(i)
```

**Output**

```
1
2
3
4
5
6
7
8
9
10
```

**Example:****Program**

```
class Counter:
    def __init__(self,n,m):
        self.min=n
        self.max=m

    def __iter__(self):
        self.ctr=self.min
        return self

    def __next__(self):
        x=self.ctr
        if self.ctr>self.max:
            raise StopIteration
        self.ctr=self.ctr+1
        return x

for i in Counter(5,10):
    print(i)
```

**Output**

```
5
6
7
8
9
10
```

**Example:****Program**

```
class Fibonacci:  
    def __init__(self,n):  
        self.n=n  
    def __iter__(self):  
        self.a=0  
        self.b=1  
        self.ctr=0  
        return self  
    def __next__(self):  
        if self.ctr==self.n:  
            raise StopIteration  
        else:  
            x=self.a  
            c=self.a+self.b  
            self.a=self.b  
            self.b=c  
            self.ctr=self.ctr+1  
            return x  
  
lst=list(Fibonacci(10))  
print(lst)
```

**Output**

```
[0, 1, 1, 2, 3, 5, 8, 13, 21,  
34]
```

# Generators

Python generator is one of the most useful and special python function ever. We can turn a function to behave as an iterator using python generators. There is a lot of overhead in building an iterator in Python; we have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, raise `StopIteration` when there was no values to be returned etc.

This is both lengthy and counter intuitive. Generator comes into rescue in such situations. Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

## Yield Statement

It works like return statement because it returns a value. The difference is that it saves the state of the function. The next time the function is called, execution continues from where it left off. It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement. If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

### Syntax:

```
def generator-func():
    statements
    -----
    yield value
```

**Example:****Program**

```
def even():
    for i in range(2,11,2):
        yield i

a=even()
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
```

**Output**

```
2
4
6
8
10
StopIteration Exception
```

**Example:****Program**

```
def Fibonacci(n):
    a=0
    b=1
    for i in range(1,n+1):
        x=a
        c=a+b
        a=b
        b=c
        yield x

for i in Fibonacci(15):
    print(i)

lst=list(Fibonacci(10))
print(lst)
```

**Output**

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
[0, 1, 1, 2, 3, 5, 8, 13, 21,
34]
```

# Itertools module

The Python itertools module is a collection of tools for handling iterators. Simply put, iterators are data types that can be used in a for loop.

## itertools functions

This module is a collection of functions.

Function	Description
<b>accumulate(iter, func)</b>	The accumulate() function takes a function as an argument. It also takes an iterable. It returns the accumulated results. The results are themselves contained in an iterable.
<b>combinations(iter, r)</b>	This function takes an iterable and a integer. This will create all the unique combination that have r members.
<b>combinations_with_replacement(iter, r)</b>	This one is just like the combinations() function, but this one allows individual elements to be repeated more than once.
<b>count(start=0, step=1)</b>	Makes an iterator that returns evenly spaced values starting with number start.
<b>cycle(iterable)</b>	It return an iterator which retrieves items from iterable in cyclic fashion when next method is call.
<b>chain(*iterables)</b>	It make an iterator that returns elements from all iterables in a sequence.
<b>compress(data, selectors)</b>	This function filters one iterable with another.
<b>dropwhile(predicate, iterable)</b>	Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.
<b>groupby(iterable, key=None)</b>	Make an iterator that returns consecutive keys and groups from the iterable. The key is a function computing a key value for each element.
<b>islice(iterable, start, stop[, step])</b>	This function is very much like slices. This function allows you to cut out a piece of an iterable.
<b>repeat(object[, times])</b>	This function will repeat an object over and over again. Unless, there is a times argument.
<b>zip(*iterables)</b>	Make an iterator that aggregates elements from each of the iterables. Note : it is not part of module itertools.
<b>zip_longest(*iterables)</b>	Make an iterator that aggregates elements from each of the iterables. Iteration continues until the longest iterable is exhausted.

**Example:****Program**

```

from itertools import *

def mul(a,b):
    return a*b

data=[1,2,3,4,5]
lst=accumulate(data,mul)

for i in lst:
    print(i)

-----
data=["red","yellow","blue"]
lst=combinations(data,2)

for i in lst:
    print(i)

lst=combinations_with_replacement(data,2)

for i in lst:
    print(i)

-----
for i in count(10,3):
    print(i)
    if i > 20:
        break

-----
data=["red","yellow","blue"]
for i in cycle(data):
    print(i)

```

**Output**

1  
2  
6  
24  
120

('red', 'yellow')  
('red', 'blue')  
(('yellow', 'blue'))  
  
(('red', 'red'))  
(('red', 'yellow'))  
(('red', 'blue'))  
(('yellow', 'yellow'))  
(('yellow', 'blue'))  
(('blue', 'blue'))

10  
13  
16  
19  
22

red  
yellow  
blue  
red  
yellow  
...

```

-----
data=["red","yellow","blue"]
lst=["Amit","Sumit"]

for i in chain(data,lst):
    print(i)

-----
data=["red","yellow","blue"]
lst=[True,False,True]

for i in compress(data,lst):
    print(i)

-----
data=[2,4,10,5,7,1,9]

for i in dropwhile(lambda x: x<=5,data):
    print(i)

-----
data=[{'Name':'Amit','result':'Pass'},
      {'Name':'Gopal','result':'fail'},
      {'Name':'Raj','result':'Pass'}
     ]

for key,group in groupby(data,lambda x:
x["result"]=="Pass"):
    if key:
        print(list(group))

-----
data=["red","yellow","blue","orange"]

for i in islice(data,2):
    print(i)

```

red  
yellow  
blue  
Amit  
Sumit

Red  
Blue

10  
5  
7  
1  
9

[{'Name': 'Amit', 'result': 'Pass'}]  
[{'Name': 'Raj', 'result': 'Pass'}]

red  
yellow

```
-----  
data=["red","yellow","blue"]  
for i in repeat(data,2):  
    print(i)
```

```
['red', 'yellow', 'blue']  
['red', 'yellow', 'blue']
```

```
-----  
fnames=["seeta","geeta","reeta"]  
lnames=["joshi","jain","pandey","doshi"]  
for fnm,lnm in zip(fnames,lnames):  
    print(fnm,lnm)
```

```
seeta joshi  
geeta jain  
reeta pandey
```

```
-----  
fnames=["seeta","geeta","reeta"]  
lnames=["joshi","jain","pandey","doshi"]  
for fnm,lnm in zip_longest(fnames,lnames):  
    print(fnm,lnm)  
  
for fnm,lnm in  
zip_longest(fnames,lnames,fillvalue="-"):  
    print(fnm,lnm)
```

```
seeta joshi  
geeta jain  
reeta pandey  
None doshi  
seeta joshi  
geeta jain  
reeta pandey  
- doshi
```

# Regular Expressions

# Regular Expressions

A regular expression (RE or regex) is a sequence of characters which describes textual patterns. Using regular expressions we can match input data for certain patterns (aka searching), extract matching strings (filtering, splitting) as well as replace occurrences of patterns with substitutions, all with a minimum amount of code.

In Python, regular expressions are supported by the `re` module. That means that if you want to start using them in your Python scripts, you have to import this module with the help of `import`

Most programming languages have built-in support for defining and operating with regular expressions. Perl, Python & Java are some notable programming languages with first-class support for regular expressions. The standard library functions in such programming languages provide highly-performant, robust and (almost) bug-free implementations of the regular expression operations (searching, filtering, etc.) that makes it easy to rapidly produce high-quality applications that process text efficiently.

## ^ (caret)

The caret symbol specifies the position for the match, at the start of the string, except when used inside square braces.

### Example:

#### Program

```
from re import *
reg="^CCIT "
msg="CCIT Amravati"
if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

#### Output

Search successful

## \$ (dollar)

The dollar symbol \$ is used to check if a string ends with a certain character.

### Example:

#### Program

```
from re import *
reg="CCIT$"
msg="Welcome to CCIT"
if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

#### Output

```
Search successful
```

## . (period)

A period matches any single character (except newline '\n').

### Example:

#### Program

```
from re import *
reg="C..T"
msg="Welcome to CCIT"
if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

#### Output

```
Search successful
```

## [] (Square brackets)

Square brackets specifies a set of characters you wish to match. The square braces match any single character enclosed within it.

- [abc] will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using - inside square brackets.

- [a-d] is the same as [abcd].
- [A-Z] Matches any character from A to Z.
- [1-4] is the same as [1234].
- [0-9] Matches any digit from 0 to 9.
- [a-zA-Z0-9] Matches any letter from (a to z) or (A to Z) or (0 to 9).

You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

- [^abc] means any character except a or b or c.
- [^0-9] means any non-digit character.

Expression	String	Result
[abc]	aabcc	True
[0-9]	2350	True
[XYZ]	xyz	False
[a-zA-Z0-9]	abs25	True

### Example:

#### Program

```
from re import *
reg="CCIT$"
msg="Welcome to CCIT"
if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

#### Output

Search successful

## \* (star)

The star symbol \* matches zero or more occurrences of the pattern left to it.

Expression	String	Result
ma*n	mn	True
	man	True
	maan	True
	main	False (a is not followed by n)
	woman	True

### Example:

#### Program

```
from re import *
reg="C*IT"
msg="CCIT"

if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

#### Output

Search successful

## + (plus)

The plus symbol + matches one or more occurrences of the pattern left to it.

Expression	String	Result
ma+n	mn	False
	man	True
	maan	True
	main	False (a is not followed by n)
	woman	True

**Example:****Program**

```
from re import *
reg="C+IT"
msg="CCIT"

if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

**Output**

Search successful

**? (Question Mark)**

The question mark symbol ? matches zero or one occurrence of the pattern left to it.

<b>Expression</b>	<b>String</b>	<b>Result</b>
ma?n	mn	False
	man	True
	maan	True
	main	False (a is not followed by n)
	woman	True

**Example:****Program**

```
from re import *
reg="C?IT"
msg="CCIT"

if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

**Output**

Search unsuccessful

## { } (Braces)

The curly braces specify the preceding element to be matched exactly n times.

- **{x}** - Repeat exactly x number of times.
- **{x,}** - Repeat at least x times or more.
- **{,y}** - Repeat at most y times or more.
- **{x, y}** - Repeat at least x times but no more than y times.

Expression	String	Result
[a-z]{4}	ccit	True
[a-z]{4}	python	False
[0-9]{2,3}	5210	False
[a-zA-Z0-9]{5,}	abs25	True

### Example:

#### Program

```
from re import *
reg="[A-Z]{4}"
msg="CCIT"
if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

#### Output

Search successful

## | (Alternation)

The vertical bar is used to separate alternatives.

Expression	String	Result
[a-zA-Z]{4} [0-9]{4}	ccit	True
[a-zA-Z]{4} [0-9]{4}	3242	True
[a-zA-Z]{4} [0-9]{4}	Python	False

**Example:****Program**

```
from re import *
reg="CCIT|ccit"
msg="CCIT"

if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

**Output**

Search successful

**() (Group)**

The vertical bar is used to separate alternatives.

<b>Expression</b>	<b>String</b>	<b>Result</b>
(P p)ython	Python	True
(P p)ython	python	True
(P p)ython	kython	False

**Example:****Program**

```
from re import *
reg="(C|c)cit"
msg="ccit"

if search(reg,msg):
    print("Search successful")
else:
    print("Search unsuccessful")
```

**Output**

Search successful

## Meta characters

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Backslash \ is used to escape various characters including all metacharacters.

Special characters	Description
\A	Matches if the specified characters are at the start of a string.
\b	Matches if the specified characters are at the beginning or end of a word.
\B	Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.
\d	Matches any decimal digit. Equivalent to [0-9]
\D	Matches any non-decimal digit. Equivalent to [^0-9]
\w	Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.
\W	Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]
\s	Matches where a string contains any whitespace character. Equivalent to [ \t\n]
\S	Matches where a string contains any non-whitespace character.
\Z	Matches if the specified characters are at the end of a string.

### Example:

#### Program

```
from re import *

reg="\w+\d*@\w+[.]\w{,3}"
msg="ccitmail@gmailcom"
if search(reg,msg):
    print("Match successfull")
else:
    print("Match unsuccessfull")
```

#### Output

Search successful

## re module Functions

The re library in Python provides several functions that makes it a skill worth mastering. This module is a collection of functions.

Function	Description
<b>findall(pattern, string, flags=0)</b>	The re.findall() method returns a list of strings containing all matches.
<b>split(pattern, string, maxsplit=0, flags=0)</b>	The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.
<b>sub(pattern, repl, string, count=0, flags=0)</b>	The method returns a string where matched occurrences are replaced with the content of replace variable.
<b>search(pattern, string, flags=0)</b>	The search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string. If the search is successful, search() returns a match object; if not, it returns None.
<b>match(pattern, string, flags=0)</b>	If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

### Example:

#### Program

```
from re import *
reg="\d+"
msg="Welcome 21 to 124 CCIT 88 Amravati"
result=findall(reg,msg)
print(result)
-----
reg="\d+"
msg="Welcome 21 to 124 CCIT 88 Amravati"
result=split(reg,msg)
print(result)
-----
```

#### Output

```
['21', '124', '88']
['Welcome ', ' to ', ' CCIT ',
 ' Amravati']
```

```
from re import *
reg="\s+"
msg="Welcome to \n CCIT Amravati"
rpl=""
result=sub(reg,rpl,msg)
print(result)
```

WelcometoCCITAmravati

## Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR ()).

Flags	Description
<code>re.I</code>	Performs case-insensitive matching.
<code>re.U</code>	Interprets letters according to the Unicode character set.

### Example:

#### Program

```
from re import *
reg="ccit"
msg="Welcome to CCIT Amravati"
result=search(reg,msg,I)
if result:
    print("search successful")
else:
    print("search unsuccessful")
```

#### Output

search successful

# Match object

This object contains information about search result. You can get methods and attributes of a match object using dir() function.

## Match object Attribute

This object contains attributes.

Attribute	Description
<code>match.re</code>	The re attribute of a matched object returns a regular expression object.
<code>match.string</code>	The string attribute returns the passed string.

## Match object method

This object contains method.

method	Description
<code>match.group([group1, ...])</code>	The group() method returns the part of the string where there is a match.
<code>match.start()</code>	The start() function returns the index of the start of the matched substring.
<code>match.end()</code>	The end() returns the end index of the matched substring.
<code>match.span()</code>	The span() function returns a tuple containing start and end index of the matched part.

**Example:****Program**

```
from re import *
reg="CCIT"
msg="Welcome to \n CCIT Amravati"
result=search(reg,msg)
print(result)

-----
reg="CCIT"
msg="CCIT Amravati"
result=match(reg,msg)
print(result)

-----
reg="CCIT"
msg="CCIT Amravati"
result=match(reg,msg)

print(result.re)
print(result.string)
print(result.start())
print(result.end())
print(result.span())
```

**Output**

```
<re.Match object; span=(13, 17), match='CCIT'>
```

```
<re.Match object; span=(0, 4), match='CCIT'>
```

```
re.compile('CCIT')
```

```
CCIT Amravati
```

```
0
```

```
4
```

```
(0, 4)
```

# Database

---

---

# Database

A database is an abstraction over an operating system's file system that makes it easier for developers to build applications that create, read, update and delete persistent data. At a high level web applications store data and present it to users in a useful way. Databases make structured storage reliable and fast. They also give you a mental framework for how the data should be saved and retrieved instead of having to figure out what to do with the data every time you build a new application.

The database storage abstraction most commonly used in Python web development is sets of relational tables. Alternative storage abstractions are explained on the NoSQL page.

Relational databases store data in a series of tables. Interconnections between the tables are specified as foreign keys. A foreign key is a unique reference from one row in a relational table to another row in a table, which can be the same table but is most commonly a different table.

Python provides us different connectors for different databases. Like

- SQLite
- MySQL
- PostgreSQL
- Oracle

## SQLite

SQLite is a database that is stored in a single file on disk. SQLite is built into Python but is only built for access by a single connection at a time. Python installation contains a Python SQL library named sqlite3 that you can use to interact with an SQLite database.



## MySQL

MySQL is another viable open source database implementation for Python applications. Python needs a MySQL driver to communicate with MySQL server. For you have to Download MySQL Driver/Connector. To Install MySQL Connector use PIP tool.



### Syntax:

```
pip install mysql-connector-python
```

## PostgreSQL

PostgreSQL is often viewed as more feature robust and stable. To work with relational databases in Python you need to use a database driver, which is also referred to as a database connector. The most common driver library for working with PostgreSQL is psycopg2.



### Syntax:

```
pip install psycopg2
```

## Oracle

Oracle database is a relational database management system. It is known as Oracle database, OracleDB or simply Oracle. Python needs a Oracle driver to communicate with Oracle server. For you have to Download Oracle Driver/Connector. To Install Oracle Connector use PIP tool.



### Syntax:

```
pip install cx_Oracle
```

# SQLite Connection

SQLite3 can be integrated with Python using sqlite3 module, which was written by Gerhard Haring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249. You do not need to install this module separately because it is shipped by default along with Python.

To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.

This API opens a connection to the SQLite database file. To do this you can use `connect` function Will connect with database and return a Connection Object if successful. When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0.

If the given database name does not exist then this call will create the database. You can specify filename with the required path as well if you want to create a database anywhere else except in the current directory.

## Syntax:

```
sqlite3.connect(database [,timeout ,other optional arguments])
```

## SQLite Connection object method

The Connection object provide many methods to perform opration on database.

Method	Description
<b>connection.close()</b>	This method closes the database connection.
<b>connection.commit()</b>	This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections.
<b>connection.rollback()</b>	This method rolls back any changes to the database since the last call to commit().
<b>connection.cursor()</b>	This routine creates a cursor which will be used throughout of your database programming with Python.

**Example:****Program**

```

import sqlite3
from sqlite3 import Error

try:
    connection = sqlite3.connect("ccitdb.db")
    print("Connected successful")
    connection.close()

except Error as e:
    print(f"Error :{e}")

```

**Output**

**Connected successful**

## SQLite Cursor object method

The Cursor object provide many methods to perform operation on database.

Method	Description
<b>cursor.execute(sql, optional parameters)</b>	This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style).
<b>cursor.executemany(sql, seq_of_parameters)</b>	This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
<b>cursor.executescript(sql_script)</b>	This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by a semi colon (:).
<b>cursor.fetchone()</b>	This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
<b>cursor.fetchmany(size = cursor.arraysize)</b>	This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
<b>cursor.fetchall()</b>	This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

## Create Table

To create table in sqlite you can use connection object method execute.

### Example:

#### Program

```
import sqlite3
from sqlite3 import Error

try:
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    query= "create table account(accno integer primary key ,name varchar,
balance int)"

    cursor.execute(query)
    connection.commit()
    print("Table created")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

#### Output

```
Table created successful
```

## Insert record Table

To insert a racode into table in sqlite.

### Example:

#### Program

```
import sqlite3
from sqlite3 import Error
try:
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    cursor.execute("insert into account values(1005,'Gopal pandey',54000);")
    connection.commit()
    print("data saved...")
    connection.close()
except Error as e:
    print(f"Error :{e}")
```

#### Output

```
data saved...
```

### Example:

**WAP to read accno, name and balance and add a record into account table.**

#### Program

```
import sqlite3
from sqlite3 import Error

try:
    acn=input("Enter Account No.")
    nam=input("Enter Name ")
    bal=input("Enter Balance ")
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    cursor.execute(f"insert into account values({acn},{nam},{bal});")
    connection.commit()
    print("data saved...")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

## Output

```
Enter Account No. 1001
Enter Name Amit Jain
Enter Balance 43000
data saved...
```

## Update record in Table

To update record in sqlite.

### Example:

#### Program

```
import sqlite3
from sqlite3 import Error

try:
    acn=input("Enter Account No.")
    bal=input("Enter Amount")
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    cursor.execute(f"update account set balance=balance+{bal} where accno={acn}")
    connection.commit()
    print("data updated")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

## Output

```
Enter Account No. 1001
Enter Amount 5000
data updated
```

## Delete record in Table

To delete record in sqlite.

### Example:

#### Program

```
import sqlite3
from sqlite3 import Error

try:
    acn=input("Enter Account No.")
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    cursor.execute(f"delete from account where accno={acn}")
    connection.commit()
    print("record removed")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

#### Output

```
Enter Account No. 1001
```

```
Record removed
```

## Retrieve one record

To retrieve one record from table in sqlite.

**Example:****Program**

```
import sqlite3
from sqlite3 import Error

try:
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    cursor.execute("select * from account")
    result=cursor.fetchone()
    print(result)

except Error as e:
    print(f"Error :{e}")
```

**Output**

(1001, 'Amit Jain', 25500)

**Retrieve all record**

To retrieve all record from table in sqlite.

**Example:****Program**

```
import sqlite3
from sqlite3 import Error

try:
    connection = sqlite3.connect("ccitdb.db")
    cursor = connection.cursor()
    cursor.execute("select * from account")
    result=cursor.fetchall()
    print(result)

except Error as e:
    print(f"Error :{e}")
```

**Output**

[(1001, 'Amit Jain', 25500), (1003, 'Mona Mantri', 15000), (1004, 'raj joshi', 24000), (1005, 'Gopal pandey', 54000)]

# MySQL Connection

MySQL is a viable open source database implementation for Python web applications. Accessing MySQL from a Python application requires a database driver (also called a "connector"). While it is possible to write a driver as part of your application, in practice most developers use an existing open source driver.

MySQL Connector is Oracle's "official" (Oracle currently owns MySQL) Python connector. The driver supports Python 2 and 3, just make sure to check the version guide for what releases work with which Python versions.

MySQL Connector/Python enables Python programs to access MySQL databases, using an API that is compliant with the Python Database API Specification v2.0 (PEP 249). It is written in pure Python and does not have any dependencies except for the Python Standard Library.

MySQL Connector/Python provide class to connect MySQL database. This class is used to manage session with database. It is defined in module `mysql.connector`

## Syntax:

```
mysql.connector.connect(host, user, password, database)
```

## MySQL Connection object method

The Connection object provide many methods to perform opration on database.

Method	Description
<b>connection.close()</b>	This method closes the database connection.
<b>connection.commit()</b>	This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections.
<b>connection.rollback()</b>	This method rolls back any changes to the database since the last call to commit().
<b>connection.cursor()</b>	This routine creates a cursor which will be used throughout of your database programming with Python.
<b>connection.is_connected()</b>	It returns True when the connection is available, False otherwise

**Example:****Program**

```

import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(host="localhost", user="root",
password="admin", database="ccitdb")
    if connection.is_connected():
        print("connection successful")
    else:
        print("connection unsuccessful")
    connection.close()
except Error as e:
    print(f"Error :{e}")

```

**Output**

**Connected successful**

## MySQL Cursor object method

The Cursor object provide many methods to perform operation on database. It is used to send a sql statement to database. MySQL cursor is read-only and non-scrollable.

Read-only: you cannot update data in the underlying table through the cursor.

Non-scrollable: you can only fetch rows in the order determined by the SELECT statement. You cannot fetch rows in the reversed order. In addition, you cannot skip rows or jump to a specific row in the result set.

Method	Description
<b>cursor.execute(sql, optional parameters)</b>	This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style).

<b>cursor.executemany(sql, seq_of_parameters)</b>	This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
<b>cursor.executescript(sql_script)</b>	This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by a semi colon (:).
<b>cursor.fetchone()</b>	This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
<b>cursor.fetchmany(size = cursor.arraysize)</b>	This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.
<b>cursor.fetchall()</b>	This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

## MySQL Cursor object Properties

The Cursor object provide Properties to perform opration on database.

Properties	Description
<b>rowcount</b>	Indicates no of rows affected
<b>with_rows</b>	returns True or False to indicate whether the most recently executed operation produced rows.

## Create Table

To create table in sqlite you can use connection object method execute.

### Example:

#### Program

```
import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    query= "create table account(accno integer primary key ,name varchar,
balance int)"
    cursor.execute(query)
    connection.commit()
    print("Table created")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

#### Output

```
Table created successful
```

## Insert record Table

To insert a racode into table in sqlite.

### Example:

#### Program

```
import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    cursor.execute("insert into account values(1005,'Gopal pandey',54000);")
    connection.commit()
    print("data saved...")
    connection.close()
except Error as e:
    print(f"Error :{e}")
```

#### Output

```
data saved...
```

**Example:****WAP to read accno, name and balance and add a record into account table.****Program**

```
import mysql.connector
from mysql.connector import Error

try:
    acn=input("Enter Account No.")
    nam=input("Enter Name ")
    bal=input("Enter Balance ")
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    cursor.execute(f"insert into account values({acn},{nam},{bal});")
    connection.commit()
    print("data saved...")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

**Output**

```
Enter Account No. 1001
Enter Name Amit Jain
Enter Balance 43000
data saved...
```

## Update record in Table

To update record in sqlite.

### Example:

#### Program

```
import mysql.connector
from mysql.connector import Error

try:
    acn=input("Enter Account No.")
    bal=input("Enter Amount")
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    cursor.execute(f"update account set balance=balance+{bal} where accno={acn}")
    connection.commit()
    print("data updated")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

#### Output

```
Enter Account No. 1001
Enter Amount 5000
data updated
```

## Delete record in Table

To delete record in sqlite.

### Example:

#### Program

```
import mysql.connector
from mysql.connector import Error

try:
    acn=input("Enter Account No.")
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    cursor.execute(f"delete from account where accno={acn}")
    connection.commit()
    print("record removed")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

#### Output

```
Enter Account No. 1001
```

```
Record removed
```

-

## Retrieve one record

To retrieve one record from table in sqlite.

### Example:

#### Program

```
import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(host="localhost", user="root",
password="admin", database="ccitdb")
    cursor = connection.cursor()
    cursor.execute("select * from account")
    result=cursor.fetchone()
    print(result)

except Error as e:
    print(f"Error :{e}")
```

#### Output

```
(1001, 'Amit Jain', 25500)
```

## Retrieve all record

To retrieve all record from table in sqlite.

**Example:**

Program

```
import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(host="localhost", user="root",
password="admin", database="ccitdb")
    cursor = connection.cursor()
    cursor.execute("select * from account")
    result=cursor.fetchall()
    print(result)
except Error as e:
    print(f"Error :{e}")
```

Output

```
[(1001, 'Amit Jain', 25500),
(1003, 'Mona Mantri', 15000),
(1004, 'raj joshi', 24000),
(1005, 'Gopal pandey', 54000)]
```

# Parameterized Statement

A Parameterized Statement is a query in which placeholders are used instead of values . The parameter values are supplied at execution time. To set placeholders use %s.

Note : parameter values can be pass in list while calling execute method of cursor object.

## Example:

### Program

```
import mysql.connector
from mysql.connector import Error

try:
    an=input("Enter AccNo: ")
    nm=input("Enter Name: ")
    bl=input("Enter Balance: ")
    lst=[an,nm,bl]
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    cursor.execute("insert into account values (%s,%s,%s)",lst)
    print("Data saved")
    connection.commit()
except Error as e:
    print(f"Error :{e}")
```

### Output

```
Enter AccNo: 1002
Enter Name: Gopal Pandey
Enter Balance: 25000
Data saved
```

**Example:****Wap to read accno and amt perform deposit operation on accmaster table.****Program**

```
import mysql.connector
from mysql.connector import Error

try:
    an=input("Enter AccNo:")
    amt=input("Enter Amount:")
    cmd="update account set balance=balance + %s where accno=%s"
    values=[amt,an]
    connection = mysql.connector.connect(host="localhost",user="root",
password="admin",database="ccitdb")
    cursor = connection.cursor()
    cursor.execute(cmd,values)
    connection.commit()
    print("Amount deposited ")
    connection.close()

except Error as e:
    print(f"Error :{e}")
```

**Output**

```
Enter Account No. 1001
Enter Amount 3000
Amount deposited
```

# Datetime

---

---

# Datetime

The datetime module provides us different classes used to perform operations on date and time objects. In Python, date, time and datetime classes provides a number of function to deal with dates, times and time intervals. Date and datetime are an object in Python, so when you manipulate them, you are actually manipulating objects and not string or timestamps. Whenever you manipulate dates or time, you need to import datetime module.

The datetime classes in Python are categorized into main 4 classes.

- date – Manipulate just date ( Month, day, year)
- time – Time independent of the day (Hour, minute, second, microsecond)
- datetime – Combination of time and date (Month, day, year, hour, second, microsecond)
- timedelta— A duration of time used for manipulating dates

## Date class

Instances of the date class represent a date (no time of day in particular within that date), are always naive, and assume the Gregorian calendar was always in effect. date instances have three read-only integer attributes: year, month, and day.

### Syntax:

```
date(year ,month ,day )
```

All arguments are required. Arguments must be integers, in the following ranges:

- $1 \leq \text{year} \leq 9999$
- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq \text{number of days in the given month and year}$

If an argument outside those ranges is given, ValueError is raised.

**Example:****Program**

```
import datetime
date=datetime.date(2020,1,25)
print(date)
```

**Output**

2020-01-25

**date class methods**

all class methods.

<b>Method</b>	<b>Description</b>
<b>date.today()</b>	Returns a date object representing today's date.
<b>date.fromisoformat( date_string)</b>	Return a date corresponding to a date_string given in the format YYYY-MM-DD
<b>date.fromisocalendar(year, week, day)</b>	Return a date corresponding to the ISO calendar date specified by year, week and day.

**Example:****Program**

```
import datetime
date=datetime.date.today()
print(date)
-----
date=datetime.date.fromisoformat("2020-01-23")
print(date)
-----
date=datetime.date.fromisocalendar(2020,3,2)
print(date)
```

**Output**

2020-04-02

2020-01-23

2020-01-14

## date class attribute

all class attribute

attribute	Description
<b>date.day</b>	Between 1 and 9999 inclusive.
<b>date.month</b>	Between 1 and 12 inclusive.
<b>date.year</b>	Between 1 and the number of days in the given month of the given year.

**Example:**

### Program

```
import datetime

date=datetime.date.today()

print(date.day)

print(date.month)

print(date.year)
```

### Output

```
15
2
2020
```

## date object method

all Instance methods

method	Description
<b>date.replace(year=self.year, month=self.month, day=self.day)</b>	Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified.
<b>d.ctime()</b>	Returns a string representing the date d in the same 24-character format as time.ctime (with the time of day set to 00:00:00, midnight).
<b>date.isocalendar()</b>	Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday). See the ISO 8601 standard for more details about the ISO (International Standards Organization) calendar.

<b>date.isoformat()</b>	Returns a string representing date object in the format 'YYYY-MM-DD'; same as str(object).
<b>date.isoweekday()</b>	Returns the day of the week of date object as an integer, 1 for Monday through 7 for Sunday; like d.weekday() + 1.
<b>date.strftime()</b>	Returns a string representing date object as specified by string fmt, like: time.strftime(fmt, object.timetuple())
<b>date.timetuple()</b>	Returns a time tuple corresponding to date object at time 00:00:00 (midnight).
<b>date.toordinal()</b>	Returns the proleptic Gregorian ordinal for date object. For example: date(1,1,1).toordinal() == 1
<b>date.weekday()</b>	Returns the day of the week of date object as an integer, 0 for Monday through 6 for Sunday; like date.isoweekday() - 1.

**Example:****Program**

```
import datetime

date=datetime.date.today()
print(date)

-----
print(date.replace(2019))

-----
print(date.ctime())

-----
print(date.isocalendar())

-----
print(date.isoformat())

-----
print(date.isoweekday())

-----
print(date.strftime("%d/%m/%Y"))

-----
print(date.timetuple())

-----
print(date.toordinal())

-----
print(date.weekday())
```

**Output**

```
2020-04-02
2019-04-02
Thu Apr  2 00:00:00 2020
(2020, 14, 4)
2020-04-02
4
02/04/2020
time.struct_time(tm_year=2020,
tm_mon=4, tm_mday=2,
tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=93,
tm_isdst=-1)
737517
3
```

# Time class

Instances of the time class represent a time of day (of no particular date), may be naive or aware regarding time zones, and always ignore leap seconds. They have five attributes: four read-only integers (hour, minute, second, and microsecond).

## Syntax:

```
time(hour=0,minute=0,second=0,microsecond=0)
```

All arguments are optional. The remaining arguments must be integers in the following ranges:

- $0 \leq \text{hour} < 24$ ,
- $0 \leq \text{minute} < 60$ ,
- $0 \leq \text{second} < 60$ ,
- $0 \leq \text{microsecond} < 1000000$

## Example:

### Program

```
import datetime

data=datetime.time(5,24,40)
print(data)
```

### Output

05:24:40

## time class attribute

all class attribute

Attribute	Description
<b>time.Hour</b>	In range(24).
<b>time.Minute</b>	In range(60).
<b>time.Second</b>	In range(60).
<b>time.Microsecond</b>	In range(1000000)

**Example:****Program**

```
import datetime

data=datetime.time(5,24,40)
print(data.hour)
print(data.minute)
print(data.second)
print(data.microsecond)
```

**Output**

5  
24  
40  
0

**time class methods**

all class methods.

<b>Method</b>	<b>Description</b>
<code>time.fromisoformat()</code>	Return a time corresponding to a time_string

**Example:****Program**

```
import datetime

data=datetime.time(5,24,40)
print(data.fromisoformat("12:30:15"))
```

**Output**

12:30:15

**date object method**

all Instance methods

<b>Method</b>	<b>Description</b>
<code>time.replace(hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond)</code>	Returns a new time object, like t except for those attributes explicitly specified as arguments, which get replaced. For example: <code>time(x,y,z).replace(minute=m) == time(x,m,z)</code>
<code>time.isoformat(timespec='auto')</code>	Returns a string representing time t in the format 'HH:MM:SS'; same as str(t).
<code>time.strftime(format)</code>	Return a string representing the time, controlled by an explicit format string.

**Example:****Program**

```
import datetime

data=datetime.time(5,24,40)
print(data.replace(2,30,50))
print(data.isoformat())
print(data.strftime("%H:%M:%S %p"))
```

**Output**

```
05:30:50
05:24:40
05:24:40 AM
```

## datetime class

Instances of the datetime class represent an instant (a date, with a specific time of day within that date), may be naive or aware of time zones, and always ignore leap seconds. datetime extends date and adds time's attributes; its instances have readonly integers year, month, day, hour, minute, second, and microsecond.

Instances of datetime support some arithmetic: the difference between datetime instances (both aware, or both naive) is a timedelta instance, and you can add or subtract a timedelta instance to/from a datetime instance to construct another datetime instance. You can compare two instances of the datetime class (the later one is greater) as long as they're both aware or both naive..

**Syntax:**

```
datetime(year,month,day,hour=0,minute=0,second=0,microsecond=0)
```

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments must be integers in the following ranges:

- 0 <= year <=9999,
- 1 <= month <= 12,
- 1 <= day <= number of days in the given month and year,
- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000

**Example:**

## Program

```
import datetime

data=datetime.datetime(2020,4,2,10,30,15)
print(data)
```

## Output

2020-04-02 10:30:15

**datetime class attribute**

all class attribute

Attribute	Description
<code>datetime.year</code>	Between 0 and 9999 inclusive.
<code>datetime.month</code>	Between 1 and 12 inclusive.
<code>datetime.day</code>	Between 1 and the number of days in the given month of the given year.
<code>datetime.hour</code>	In range(24).
<code>datetime.minute</code>	In range(60).
<code>datetime.second</code>	In range(60).
<code>datetime.microsecond</code>	In range(1000000).

**Example:**

## Program

```
import datetime

data=datetime.datetime(2020,4,2,10,30,15)

print(data.year)
print(data.month)
print(data.day)
print(data.hour)
print(data.minute)
print(data.second)
print(data.microsecond)
```

## Output

2020  
4  
2  
10  
30  
15  
0

## datetime class methods

all class methods.

Method	Description
<code>datetime.combine(date, time)</code>	Returns a datetime object with the date attributes taken from date and the time attributes (including tzinfo) taken from time. <code>datetime.combine(d,t)</code>
<code>datetime.now()</code>	Returns a datetime object for the current local date and time
<code>datetime.today()</code>	Returns a naive datetime object representing the current local date and time, same as the now class method
<code>datetime.strptime(str, fmt='%a %b %d %H:%M:%S %Y')</code>	Returns a datetime representing str as specified by string fmt.
<code>datetime.fromisoformat(date_string)</code>	Return a datetime corresponding to a date_string
<code>datetime.fromisocalendar(year, week, day)</code>	Return a datetime corresponding to the ISO calendar date specified by year, week and day.

**Example:**

### Program

```
import datetime

date=datetime.date.today()
time=datetime.time(2,30,45)
print(datetime.datetime.combine(date,time))
print(datetime.datetime.now())
print(datetime.datetime.today())
print(datetime.datetime.strptime("2020/4/3
6:30:45","%Y/%m/%d %H:%M:%S"))
print(datetime.datetime.fromisoformat("2020-
12-05 05:15:48"))
print(datetime.datetime.fromisocalendar(2020,
24,3))
```

### Output

```
2020-04-03 02:30:45
2020-04-03 15:31:51.880448
2020-04-03 15:31:51.892443
2020-04-03 06:30:45
2020-12-05 05:15:48
2020-06-10 00:00:00
```

## date object method

all Instance methods

Method	Description
<code>datetime.replace( year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond)</code>	Returns a new datetime object, like d except for those attributes specified as arguments, which get replaced.
<code>datetime.date()</code>	Returns a date object representing the same date as datetime.
<code>datetime.time()</code>	Returns a naive time object representing the same time of day as datetime.
<code>datetime.ctime()</code>	Returns a string representing date and time d in the same 24-character format as time.ctime.
<code>datetime.isocalendar()</code>	Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday) for date.
<code>datetime.isoformat(sep='T')</code>	Returns a string representing d in the format 'YYYY-MM-DDxHH:MM:SS', where x is the value of argument sep (must be a string of length 1).
<code>datetime.isoweekday()</code>	Returns the day of the week of d's date as an integer; 1 for Monday through 7 for Sunday.
<code>datetime.weekday()</code>	Returns the day of the week of d's date as an integer; 0 for Monday through 6 for Sunday.
<code>datetime.strftime(format)</code>	Returns a string representing d as specified by the format string format.

**Example:****Program**

```
import datetime

data=datetime.datetime.now()

print(data)

print(data.replace(2014,5,8))

print(data.date())

print(data.time())

print(data.ctime())

print(data.isocalendar())

print(data.isoformat())

print(data.isoweekday())

print(data.isoweekday())

print(data.strftime("%d/%m/%Y %H:%M:%S " ))
```

**Output**

2020-04-03 22:35:39.028798  
2014-05-08 22:35:39.028798  
2020-04-03  
22:35:39.028798  
Fri Apr 3 22:35:39 2020  
(2020, 14, 5)  
2020-04-03T22:35:39.028798  
5  
5  
03/04/2020 22:35:39

## Format Codes

date, datetime, and time objects all support a strftime(format) method, to create a string representing the time under the control of an explicit format string.

Conversely, the datetime.strptime() class method creates a datetime object from a string representing a date and time and a corresponding format string..

<b>Directive</b>	<b>Description</b>	<b>Example</b>
%a	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat

<b>%A</b>	Weekday as locale's full name.	Sunday, Monday, ..., Saturday
<b>%w</b>	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0,1,...,6
<b>%d</b>	Day of the month as a zero-padded decimal number.	01,02,....,31
<b>%b</b>	Month as locale's abbreviated name.	Jan, Feb, ..., Dec
<b>%B</b>	Month as locale's full name	January,February,..., December
<b>%m</b>	Month as a zero-padded decimal number.	01, 02, ..., 12
<b>%y</b>	Year without century as a zero-padded decimal number.	00, 01, ..., 99
<b>%Y</b>	Year with century as a decimal number.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
<b>%H</b>	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
<b>%I</b>	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
<b>%p</b>	Locale's equivalent of either AM or PM.	AM, PM
<b>%M</b>	Minute as a zero-padded decimal number.	00, 01, ..., 59
<b>%S</b>	Second as a zero-padded decimal number.	00, 01, ..., 59
<b>%f</b>	Microsecond as a decimal number, zero-padded on the left.	000000, 000001, ..., 999999
<b>%j</b>	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
<b>%U</b>	Week number of the year (Sunday as the first day of the week) as a zero padded decimal number.	00, 01, ..., 53
<b>%c</b>	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988
<b>%x</b>	Locale's appropriate date representation.	08/16/88
<b>%X</b>	Locale's appropriate time representation.	21:30:00

**Example:****Program**

```
import datetime

data=datetime.datetime.now()
print(data.strftime("%A %d/%B/%Y %H:%M:%S
%p"))
```

**Output**

Saturday 04/April/2020  
04:40:28 AM

## timedelta class

A timedelta object represents a duration, the difference between two dates or times.

**Syntax:**

```
timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0
, hours=0, weeks=0)
```

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

Only days, seconds and microseconds are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- $0 \leq \text{microseconds} < 1000000$
- $0 \leq \text{seconds} < 3600*24$  (the number of seconds in one day)
- $-999999999 \leq \text{days} \leq 999999999$

**Example:****Program**

```
import datetime

data=datetime.timedelta(hours=10,minutes=20,s
econds=30)
print(data)
```

**Output**

5:20:30

**timedelta class attribute**

all class attribute

<b>Attribute</b>	<b>Description</b>
<b>timedelta. days</b>	Between -999999999 and 999999999 inclusive
<b>timedelta. seconds</b>	Between 0 and 86399 inclusive
<b>timedelta. microseconds</b>	Between 0 and 999999 inclusive

**timedelta class operation**

Supported operations:

<b>Operator</b>	<b>Description</b>
<b>+</b>	Addition
<b>-</b>	Difference
<b>*</b>	Delta multiplied by an integer.
<b>/</b>	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.

**Example:****Program**

```
import datetime

t1=datetime.timedelta(hours=2,minutes=10,seconds=00)
t2=datetime.timedelta(hours=3,minutes=30,seconds=00)

result=t1+t2
print(result)

result=t2-t1
print(result)

result=t1*2
print(result)

result=t1/2
print(result)

result=t2%t1
print(result)
```

**Output**

```
5:40:00
1:20:00
4:20:00
1:05:00
1:20:00
```

**Example:****Program**

```
import datetime

t1=datetime.date(2020,2,3)
t2=datetime.timedelta(days=5)
result=t1-t2

print(result)
```

**Output**

```
2020-01-29
```

# calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year. By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call `calendar.setfirstweekday`. calendar handles years in module time's range, typically (at least) 1970 to 2038.

## calendar method

Method	Description
<b>calendar(year,w=2,l=1,c=6)</b>	Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length $21*w+18+2*c$ . l is the number of lines for each week
<b>setfirstweekday(weekday)</b>	Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience.
<b>firstweekday()</b>	Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.
<b>isleap(year)</b>	Returns True if year is a leap year; otherwise, False.
<b>leapdays(y1,y2)</b>	Returns the total number of leap days in the years within range(y1,y2) (remember, this means that y2 is excluded).
<b>month(year,month,w=2,l=1)</b>	Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length $7*w+6$ . l is the number of lines for each week.
<b>monthcalendar(year,month)</b>	Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.
<b>monthrange(year,month)</b>	Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.
<b>weekday(year,month,day)</b>	Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (Jan) to 12 (Dec).

**Example:****Program**

```
import calendar

print(calendar.calendar(2021))
```

**Output**

2021													
<b>January</b>					<b>February</b>					<b>March</b>			
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
							1	2	3	4	5	6	7
1	2	3					8	9	10	11	12	13	14
4	5	6	7	8	9	10	15	16	17	18	19	20	21
11	12	13	14	15	16	17	22	23	24	25	26	27	28
18	19	20	21	22	23	24	29	30	31				
25	26	27	28	29	30	31							
<b>April</b>					<b>May</b>					<b>June</b>			
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4				1	2					
5	6	7	8	9	10	11	3	4	5	6	7	8	9
12	13	14	15	16	17	18	10	11	12	13	14	15	16
19	20	21	22	23	24	25	17	18	19	20	21	22	23
26	27	28	29	30			24	25	26	27	28	29	30
							31						
<b>July</b>					<b>August</b>					<b>September</b>			
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4										
5	6	7	8	9	10	11	2	3	4	5	6	7	8
12	13	14	15	16	17	18	9	10	11	12	13	14	15
19	20	21	22	23	24	25	16	17	18	19	20	21	22
26	27	28	29	30	31		23	24	25	26	27	28	29
							30	31					
<b>October</b>					<b>November</b>					<b>December</b>			
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
							1	2	3	4	5	6	7
1	2	3					8	9	10	11	12	13	14
4	5	6	7	8	9	10	15	16	17	18	19	20	21
11	12	13	14	15	16	17	22	23	24	25	26	27	28
18	19	20	21	22	23	24	29	30					
25	26	27	28	29	30	31							

**Example:****Program**

```
import calendar

calendar.setfirstweekday(calendar.SATURDAY)
print(calendar.firstweekday())

print(calendar.month(2020,1))

print(calendar.isleap(2020))

print(calendar.leapdays(2020,2031))

print(calendar.monthcalendar(2021,1))

print(calendar.monthrange(2021,2))

print(calendar.weekday(2021,1,2))
```

**Output**

```
5
January 2020
Sa Su Mo Tu We Th Fr
      1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

True

3

[[0, 0, 0, 0, 1, 2, 3],
[4, 5, 6, 7, 8, 9, 10],
[11, 12, 13, 14, 15, 16, 17],
[18, 19, 20, 21, 22, 23, 24],
[25, 26, 27, 28, 29, 30, 31]]

(0, 28)

5
```

# time Module

Python has a module named time to handle time-related tasks. To use functions defined in the module, we need to import the module first. Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

## calendar method

Method	Description
<code>time.time()</code>	The time() function returns the number of seconds passed since epoch.
<code>time.ctime()</code>	The time.ctime() function takes seconds passed since epoch as an argument and returns a string representing local time.
<code>time.localtime()</code>	The localtime() function takes the number of seconds passed since epoch as an argument and returns struct_time in local time.
<code>time.asctime()</code>	Accepts a timetuple and returns a readable 24-character string such as 'Sun Jan 8 14:41:06 2017'.
<code>perf_counter()</code>	Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide.

### Example:

#### Program

```
import time

print(time.time())
print(time.ctime())
print(time.asctime())
print(time.localtime())
print(time.perf_counter())
```

#### Output

```
1586029730.6842942
Sun Apr  5 01:18:50 2020
Sun Apr  5 01:18:50 2020
time.struct_time(tm_year=2020,
tm_mon=4, tm_mday=5,
tm_hour=1, tm_min=18,
tm_sec=50, tm_wday=6,
tm_yday=96, tm_isdst=0)
0.5039166
```

## time.sleep function

Python module time which provides several useful functions to handle time-related tasks. One of the popular functions among them is sleep(). The sleep() function suspends (waits) execution of the current thread for a given number of seconds.

The sleep() function suspends execution of the current thread for a given number of seconds.

### Syntax:

```
time.sleep(seconds=value)
```

### Example:

#### Program

```
import time
import datetime

while True:
    data=datetime.datetime.now()
    print(data.time())
    time.sleep(1)
```

#### Output

```
01:29:21.661419
01:29:22.668518
01:29:23.681959
01:29:24.689058
01:29:25.702926
01:29:26.713314
01:29:27.724337
01:29:28.743706
...
..
.
```

# File I/O

---

---

# File I/O

A File is sequence of bytes stored on secondary storage. File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk). Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data. When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Before we can go into how to work with files in Python, it's important to understand what exactly a file is and how modern operating systems handle some of their aspects.

At its core, a file is a contiguous set of bytes used to store data. This data is organized in a specific format and can be anything as simple as a text file or as complicated as a program executable. In the end, these byte files are then translated into binary 1 and 0 for easier processing by the computer.

Files on most modern file systems are composed of three main parts:

- Header: metadata about the contents of the file (file name, size, type, and so on)
- Data: contents of the file as written by the creator or editor
- End of file (EOF): special character that indicates the end of the file

## Type of File

There are two separate types of files that Python handles: binary and text files. Knowing the difference between the two is important because of how they are handled.

### Text files

Text files contain plain text characters. When you open these in a text editor, they show human-readable content. These files store End of Line (EOL) marker at the end of each line to represent line break and an End of File (EOF) at the end of the file to represent end of file.

### Binary files

A binary file stores information in bytes that aren't quite so humanly readable. Binary files are those typical files that store data in the form of sequence of bytes grouped into eight bits or sometimes sixteen bits. These bits represent custom data and such files can store multiple types of data (images, audio, text, etc) under a single file.

## File Open

When you want to work with a file, the first thing to do is to open it. This is done by invoking the `open()` built-in function. This function takes two arguments. The first one is file address and the other one is opening mode. There are some modes to open a file.

Open file and return a corresponding file object. If the file cannot be opened, an `OSError` is raised. `file` is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless `closefd` is set to `False`.) `mode` is an optional string that specifies the mode in which the file is opened. It defaults to '`r`' which means open for reading in text mode.

### Syntax:

```
open(filename, mode='r')
```

### File mode

The mode in the `open` function tells Python what you want to do with the file. There are multiple modes that you can specify when dealing with text files.

Character	Meaning	During Inexistence of file
' <code>r</code> ' / ' <code>rb</code> '	Open text/binary for reading.	If the file does not exist, <code>open()</code> returns <code>None</code> .
' <code>w</code> ' / ' <code>wb</code> '	Open text/binary for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
' <code>a</code> ' / ' <code>ab</code> '	Open text/binary for append. i.e, Data is added to end of file.	If the file does not exist, it will be created.
' <code>r+</code> ' / ' <code>rb+</code> '	Open text/binary for both reading and writing.	If the file does not exist, <code>open()</code> returns <code>None</code> .
' <code>w+</code> ' / ' <code>wb+</code> '	Open text/binary for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.

## File opening ways

There are basically two ways to use the open method. With one syntax you assign a variable name to the file, and use this variable name in later code to refer to the file. That syntax is :

### Syntax:

```
var = open(filename, mode='r')
```

Replace var with a name of your choosing (though it's very common in Python to use just the file as the name). Although this method does work, it's not ideal because, after it's opened, the file remains open until you specifically close it using the close() method. Forgetting to close files can cause the problem of having too many files open at the same time, which can corrupt the contents of a file or cause your app to raise an exception and crash.

After the file is open, there are a few ways to access its content. For now, we simply copy everything that's in the file to a variable named filecontents in Python, and then we display this content using a simple print() function. So to open data.txt, read in all its content, and display that content on the screen, use this code:

### Example:

#### Program

```
file=open("data.txt")
filecontent=file.read()
print(filecontent)
file.close()
```

#### Output

```
Welcome to CCIT
```

With this method, the file remains open until you specifically close it using the file variable name and the .close() method. It's important for your apps to close any files it no longer needs open. Failure to do so allows open file handlers to accumulate, which can eventually cause the app to throw an exception and crash, perhaps even corrupting some of the open files along the way.

## With statement

The with statement is used to wrap the execution of a block with methods defined by a context manager. This method starts with the word with. You still assign a variable name. But you do so near the end of the line. The very last thing on the line is a colon which marks the beginning of the with block. All indented code below that is assumed to be relevant to the context of the open file (like code indented inside a loop). At the end of this you don't need to specifically close the file; Python does it automatically:

**Example:****Program**

```
with open("data.txt") as file:  
    filecontent=file.read()  
    print(filecontent)
```

**Output**

Welcome to CCIT

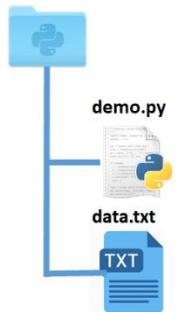
## File Methods

There are various methods available with the file object.

Method	Description
<code>file.close()</code>	Close an open file. It has no effect if the file is already closed.
<code>file.fileno()</code>	Return an integer number (file descriptor) of the file.
<code>file.read(n)</code>	Read atmost n characters form the file. Reads till end of file if it is negative or None.
<code>file.readable()</code>	Returns True if the file stream can be read from.
<code>file.readline(n=-1)</code>	Read and return one line from the file. Reads in at most n bytes if specified.
<code>file.readlines(n=-1)</code>	Read and return a list of lines from the file. Reads in at most n bytes/characters if specified.
<code>file.seek(offset, from=SEEK_SET)</code>	Change the file position to offset bytes, in reference to from (start, current, end).
<code>file.seekable()</code>	Returns True if the file stream supports random access.
<code>file.tell()</code>	Returns the current file location.
<code>file.truncate(size=None)</code>	Resize the file stream to size bytes. If size is not specified, resize to current location.
<code>file.flush()</code>	Flush the write buffer of the file stream.
<code>file.writable()</code>	Returns True if the file stream can be written to.
<code>file.write(s)</code>	Write string s to the file and return the number of characters written.
<code>file.writelines(lines)</code>	Write a list of lines to the file.

## File Read

To read data from file data.txt file.the file must be open in read mode.



### To read contents of file Example:

#### Program

```
file=open("data.txt")
filecontent=file.read()
print(filecontent)
file.close()
```

#### Output

```
Welcome to Python
at
CCIT
Amravati
```

### To read specified no of chars of file Example:

#### Program

```
file=open("data.txt")
filecontent=file.read(15)
print(filecontent)
file.close()
```

#### Output

```
Welcome to Pyth
```

### To read a line from file Example:

#### Program

```
file=open("data.txt")
filecontent=file.readline()
print(filecontent)
file.close()
```

#### Output

```
Welcome to Python
```

**To read all lines from file Example:****Program**

```
file=open("data.txt")
filecontent=file.readlines()
print(filecontent)
file.close()
```

**Output**

```
[ 'Welcome to Python\n',
  'at\n',
  'CCIT\n',
  'Amravati' ]
```

**To check file is readable or not Example:****Program**

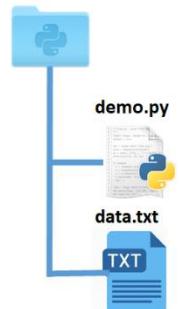
```
file=open("data.txt",'r')
if file.readable() :
    print("File is readable...")
else:
    print("File is not open in read mode")
file.close()
```

**Output**

```
File is readable...
```

**File write**

To write data to file data.txt file.the file must be open in w mode.

**To check file is writable or not Example:****Program**

```
file=open("data.txt",'w')
if file.writable() :
    print("File is writable...")
else:
    print("File is not open in write mode")
file.close()
```

**Output**

```
File is writable...
```

**To write data on file Example:****Program**

```
file=open("data.txt",'w')
file.write("Welcome to CCIT")
print("Data successfully stored ")
file.close()
```

**Output**

Data successfully stored

**To write data on file Example:****Program**

```
fileobj=open("data.txt",'w')
print("Welcome to CCIT",file=fileobj)
print("Data successfully stored")
fileobj.close()
```

**Output**

Data successfully stored

**Note :** To write data on file we can use print function. By default, print() is bound to sys.stdout through its file argument, but you can change that. Use that keyword argument to indicate a file that was open in write or append mode, so that messages go straight to it.

**To write list of data on file Example:****Program**

```
fileobj=open("data.txt",'w')
lst=['Amit Jain\n','Gopal Pandey\n','Raj
Kumar']
fileobj.writelines(lst)
print("Data successfully stored")
fileobj.close()
```

**Output**

Data successfully stored

# Serialization

Data serialization is the process of converting structured data to a format that allows sharing or storage of the data in a form that allows recovery of its original structure. In some cases, the secondary intention of data serialization is to minimize the data's size which then reduces disk space or bandwidth requirements. In serialization, an object is transformed into a format that can be stored, so as to be able to deserialize it later and recreate the original object from the serialized format.

In Python, when we want to serialize and de-serialize a Python object, we use functions and methods from the module Python Pickle. Pickling, then, is the act of converting a Python object into a byte stream. We also call this 'serialization' , 'marshalling' , or 'flattening' . Unpickling is its inverse, ie., converting a byte stream from a binary file or bytes-like object into an object.

## Pickle module

Pickle is used for serializing and de-serializing Python object structures, also called marshalling or flattening. Serialization refers to the process of converting an object in memory to a byte stream that can be stored on disk or sent over a network. Later on, this character stream can then be retrieved and de-serialized back to a Python object.

Pickling is useful for applications where you need some degree of persistency in your data. Your program's state data can be saved to disk, so you can continue working on it later on. It can also be used to send data over a Transmission Control Protocol (TCP) or socket connection, or to store python objects in a database.

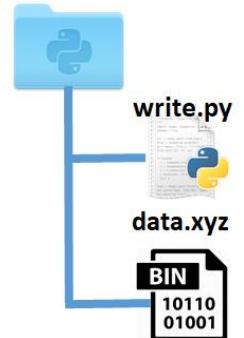
## Pickle Methods

There are various methods available for serializing and de-serializing.

Method	Description
<b>pickle.dump(value,fileobj)</b>	The dump() method serializes to an open file (file-like object). dumps returns a bytestring representing object value. dump writes the same string to the filelike object fileobj, which must be opened for writing. dump(v,f) is like f.write(dumps(v)).
<b>pickle.load(fileobj)</b>	Deserializes from an open-like object. In other words, a sequence of calls to load(f) deserializes the same values previously serialized when f's contents were created by a sequence of calls to dump(v,f). load reads the right number of bytes from file-like object fileobj and creates and returns the object v represented by those bytes.

## Write binary file

To write data on file data.xyz file.the file must be open in write binary mode.



### To write number object on binary file Example:

Program

```
import pickle
fileobj=open("data.xyz",'wb')
a=2741
pickle.dump(a,fileobj)
print("Data saved")
fileobj.close()
```

Output

Data saved

### To writen list object on binary file Example:

Program

```
import pickle
fileobj=open("data.xyz",'wb')
lst=[251,2342,"Amit",2.25]
pickle.dump(lst,fileobj)
print("Data saved")
fileobj.close()
```

Output

Data saved

### To write dictionary object on binary file Example:

Program

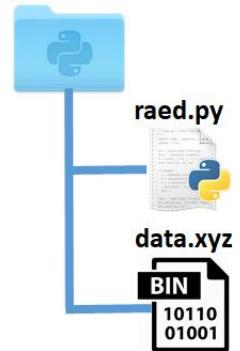
```
import pickle
fileobj=open("data.xyz",'wb')
dit={'rollno':1227,'name':"Amit Jain",'age':
35}
pickle.dump(dit,fileobj)
print("Data successfully stored")
fileobj.close()
```

Output

Data successfully stored

## Read binary file

To read data from file data.xyz file. The file must be open in read binary mode.



### To read dictionary object from binary file Example:

#### Program

```
import pickle
fileobj=open("data.xyz",'rb')
a=pickle.load(fileobj)
print("The stored value",a)
fileobj.close()
```

#### Output

The stored value 2741

### To read list object from binary file Example:

#### Program

```
import pickle
fileobj=open("data.xyz",'rb')
a=pickle.load(fileobj)
print("The stored value",a)
fileobj.close()
```

#### Output

The stored value [251, 2342, 'Amit', 2.25]

### To read dictionary object from binary file Example:

#### Program

```
import pickle
fileobj=open("data.xyz",'rb')
dit=pickle.load(fileobj)
print(dit)
fileobj.close()
```

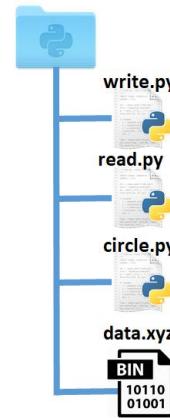
#### Output

{'rollno': 1227, 'name': 'Amit Jain', 'age': 35}

## reading and writing an object in binary file

To read and write circle object to file data.xyz file.

The file must be open in binary mode.



### To write circle object on binary file Example:

#### Program

```

circle.py
-----
class circle:
    def __init__(self,r):
        self.R=r

    def area(self):
        A=3.14*self.R**2
        print("Area is ",A)

    def circumference(self):
        C=2*3.14*self.R
        print("Circumference is ",C)

write.py
-----
import pickle
import circle

A=circle.circle(5.2)
fileobj=open("data.xyz",'wb')
pickle.dump(A,fileobj)
print("Data successful stored")
fileobj.close()

read.py
-----
import pickle
import circle

fileobj=open("data.xyz",'rb')
A=pickle.load(fileobj)
A.area()
A.circumference()
fileobj.close()
  
```

#### Output

Data successfully stored

Area is 84.9056

Circumference is 32.6560000006

# os Module

os is an umbrella module presenting a reasonably uniform cross-platform view of the capabilities of various operating systems. It supplies low-level ways to create and handle files and directories, and to create, manage, and destroy processes.

The os module supplies a name attribute, a string that identifies the kind of platform on which Python is being run. Common values for name are 'posix' (all kinds of Unix-like platforms, including Linux and macOS), 'nt' (all kinds of Windows platforms), and 'java' (Jython). You can exploit some unique capabilities of a platform through functions supplied by os. However, this book deals with crossplatform programming, not with platform-specific functionality, so we do not cover parts of os that exist only on one platform, nor platform-specific modules. Functionality covered in this book is available at least on 'posix' and 'nt' platforms. We do, though, cover some of the differences among the ways in which a given functionality is provided on various platforms.

## Filesystem Operations

Using the os module, you can manipulate the filesystem in a variety of ways: creating, copying, and deleting files and directories; comparing files; and examining filesystem information about files and directories.

### Path-String Attributes of the os Module

A file or directory is identified by a string, known as its path, whose syntax depends on the platform. On both Unix-like and Windows platforms, Python accepts Unix syntax for paths, with a slash (/) as the directory separator. On non-Unix-like platforms, Python also accepts platform-specific path syntax. On Windows, in particular, you may use a backslash (\) as the separator.

The os module supplies attributes that provide details about path strings on the current platform.

**curdir :-** The string that denotes the current directory ('.' on Unix and Windows)

**extsep :-** The string that separates the extension part of a file name from the rest of the name ('.' on Unix and Windows)

**pardir :-** The string that denotes the parent directory ('..' on Unix and Windows)

**pathsep :-** The separator between paths in lists of paths, such as those used for the environment variable PATH (':' on Unix; ';' on Windows)

**sep :-** The separator of path components ('/' on Unix; '\\\' on Windows)

## os Methods

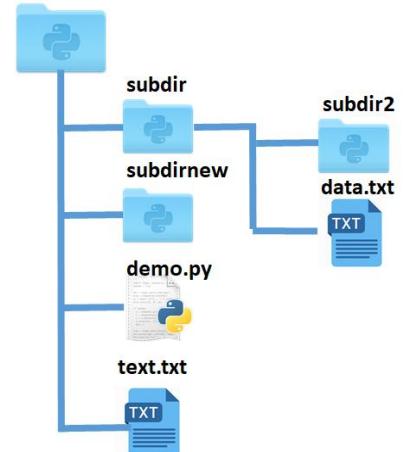
The os module supplies several functions to query and set file and directory status.

Method	Description
<b>chdir(path)</b>	Sets the current working directory of the process to path.
<b>getcwd()</b>	Returns a str, the path of the current working directory.
<b>listdir(path)</b>	Returns a list whose items are the names of all files and subdirectories in the directory path. The list is in arbitrary order and does not include the special directory names '.' (current directory) and '..' (parent directory).
<b>mkdir(path, mode)</b>	mkdir creates only the rightmost directory of path and raises OSError if any of the previous directories in path do not exist.
<b>makedirs(path, mode)</b>	makedirs creates all directories that are part of path and do not yet exist.
<b>remove(path)</b>	Removes the file named path.
<b>rmdir(path)</b>	The rmdir() method deletes the directory, which is passed as an argument in the method. Before removing a directory, all the contents in it should be removed.
<b>removedirs(path)</b>	Loops from right to left over the directories that are part of path, removing each one. The loop ends when a removal attempt raises an exception, generally because a directory is not empty.
<b>rename(source, dest)</b>	Renames (i.e., moves) the file or directory named source to dest. If dest already exists, rename may either replace dest, or raise an exception
<b>walk(top,topdown=True)</b>	Python method walk() generates the file names in a directory tree by walking the tree either top-down or bottom-up. top – Each directory rooted at directory, yields 3-tuples, i.e., (dirpath, dirnames, filenames)
<b>access(path, mode)</b>	Returns True when the file path has all of the permissions encoded in integer mode otherwise, False. <b>Mode:</b> <ul style="list-style-type: none"> <li>• os.F_OK – to test the existence of path.</li> <li>• os.R_OK – to test the readability of path.</li> <li>• os.W_OK - to test the writability of path.</li> <li>• os.X_OK - to determine if path can be executed.</li> </ul>
<b>chmod(path, mode)</b>	Changes the permissions of the file path, as encoded in integer mode.

## stat module

The stat module defines constants and functions for interpreting the results of os.stat(), os.fstat() and os.lstat() (if they exist).

Attribute	Description
<code>stat.S_IRUSR</code>	read permission.
<code>stat.S_IWUSR</code>	write permission.
<code>stat.S_IXUSR</code>	execute permission.



### Example:

#### Program

```
import os

print(os.getcwd())
-----
os.chdir("./subdir")
print(os.getcwd())
-----

print(os.listdir())
-----

os.mkdir('subdirnew')
print(os.listdir())
-----

os.makedirs('subdir/subdir2')
print(os.listdir())
os.chdir('./subdir')
print(os.listdir())
-----

print(os.listdir())
os.remove('data.txt')
print(os.listdir())
```

#### Output

```
C:\Users\CCIT\Desktop\Python

C:\Users\CCIT\Desktop\Python\subdir

['demo.py', 'subdir', 'text.txt']

['demo.py', 'subdir', 'subdirnew', 'text.txt']

['demo.py', 'subdir', 'text.txt']

['data.txt', 'subdir2']

['data.txt', 'demo.py', 'subdir', 'text.txt']

['demo.py', 'subdir', 'text.txt']
```

## Program

```

import os

print(os.listdir())
os.rmdir('subdirnew')
print(os.listdir())
-----
print(os.listdir())
os.removedirs('subdirnew/subdir2')
print(os.listdir())
-----
print(os.listdir())
os.rename('text.txt', 'data.txt')
print(os.listdir())
-----
for dirname,dirname,filename in os.walk('.'):
    print("DirPath: ",dirname)
    print("DirName: ",dirname)
    print("FileName: ",filename)

-----
print(os.access('data.txt',os.R_OK))

-----
os.chmod('data.txt',os.W_OK)
print(os.access('data.txt',os.W_OK))

```

## Output

```

['demo.py', 'subdir', 'subdirnew', 'text.txt']

['demo.py', 'subdir', 'text.txt']

['demo.py', 'subdir', 'subdirnew', 'text.txt']

['demo.py', 'subdir', 'text.txt']

['demo.py', 'subdir', 'text.txt']

['data.txt', 'demo.py', 'subdir']

DirPath: .
DirName: ['subdir', 'subdirnew']
FileName: ['data.txt', 'demo.py']

DirPath: .\subdir
DirName: ['subdir2']
FileName: ['data.txt']

DirPath: .\subdir\subdir2
DirName: []
FileName: []

DirPath: .\subdirnew
DirName: []
FileName: []

True

True

```

## os.path module

The os.path module supplies functions to analyze and transform path strings. To use this module, you can import os.path; however, even if you just import os, you can also access the os.path module and all of its attributes.

Method	Description
<b>abspath(path)</b>	Returns a normalized absolute path string equivalent to path
<b>basename(path)</b>	Returns the base name part of path
<b>dirname(path)</b>	Returns the directory part of path
<b>exists(path)</b>	Returns True when path names an existing file or directory; otherwise, False.
<b>isfile(path)</b>	Returns True when path names an existing regular file otherwise, False.
<b>isdir(path)</b>	Returns True when path names an existing directory otherwise, False.
<b>join(path, *paths)</b>	Returns a string that joins the argument strings with the appropriate path separator for the current platform.
<b>split(path)</b>	Returns a pair of strings (dir, base) such that join(dir, base) equals path. base is the last component and never contains a path separator.
<b>splitext(path)</b>	Returns a pair (root, ext) such that root+ext equals path. ext is either '' or starts with a '.' and has no other '.' or path separator.

### Example:

#### Program

```
import os

print(os.path.abspath(os.getcwd()))
-----
print(os.path.basename(os.getcwd()))
-----
print(os.path.dirname(os.getcwd()))
-----
print(os.path.exists("./demo.py"))
```

#### Output

```
C:\Users\CCIT\Desktop\Python

Python

C:\Users\CCIT\Desktop

True
```

**Program**

```
import os

print(os.path.isfile("./demo.py"))
-----
print(os.path.isdir("./demo.py"))
-----
path=os.path.join(os.getcwd(),"demo.py")
print(path)
-----
data=os.path.split(path)
print(data)
-----
data=os.path.splitext("demo.py")
print(data)
```

**Output**

```
True
False
C:\Users\CCIT\Desktop\Python\demo.py
('C:\\\\Users\\\\CCIT\\\\Desktop\\\\Python', 'demo.py')
('demo', '.py')
```

# MultiThreading

---

---

# MultiThreading

Python is a multi-threaded programming language which means we can develop multi-threaded program using Python. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

## Thread

A thread is a path of execution. A multithreaded program contains multiple paths of execution. Multithreading allows a program to remain responsive while one thread waits for input and another runs a GUI at the same time.

Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.

All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.

## Threading Module

The threading module provides all the methods of the thread module and provides some additional methods

Method	Description
<b>active_count()</b>	Returns an int, the number of Thread objects currently alive.
<b>current_thread()</b>	Returns a Thread object for the calling thread. If the calling thread was not created by threading, current_thread creates and returns a semi-dummy Thread object with limited functionality.
<b>enumerate()</b>	Returns a list of all Thread objects currently alive.

# Thread class

An object of this type represents a Thread. This class provides basic features of a thread. It provides methods to control the thread. We can derive our classes from this class to concurrently execute some code. The code which we want to concurrently execute must be placed in its run method.

## Syntax:

```
Thread(group=None, target=None, name=None, args=(), kwargs={})
```

**group** : Should be None; reserved for future extension when a ThreadGroup class is implemented.

**name** : The thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

**target** : Indicates a function name. i.e. Callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

**args** : Argument to be passed to target function.

**kwargs** : Keyword argument dictionary for the target invocation. Defaults to {}.

Note: If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.\_\_init\_\_()) before doing anything else to the thread.

## Thread object Attribute

This object contains attributes.

Attribute	Description
<b>Daemon</b>	A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before start() is called, otherwise RuntimeError is raised.
<b>Name</b>	A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

## Thread object Method

This object contains method.

Method	Description
<b>start()</b>	Start the threads activity. It must be called at most once per thread object. It arranges for the objects run() method to be invoked in a separate thread of control.
<b>run()</b>	run is the method that executes threads main function. Subclasses of Thread can override run. Unless overridden, run calls the target callable passed on t's creation.
<b>join(timeout=None)</b>	Suspends the calling thread until thread terminates (when t is already terminated, the calling thread does not suspend). You can call join only after start. It's OK to call join more than once. When the timeout argument is not present or None, the operation will block until the thread terminates.
<b>is_alive()</b>	Return whether the thread is alive. This method returns True just before the run() method starts until just after the run() method terminates.
<b>getName()</b>	It returns the name of a thread.
<b>setName(name)</b>	It sets the name of a thread.
<b>setDaemon(daemonic)</b>	Set the thread's daemon flag to the Boolean value daemonic. This must be called before start() is called. Daemon threads are automatically terminated when all user threads have completed their execution.

A thread is basically an independent flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. For Example, when you are playing a game say FIFA on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc. All these are separate threads responsible for carrying out these different tasks in the same program.

Every process has one thread that is always running. This is the main thread. This main thread actually creates the child thread objects. The child thread is also initiated by the main thread.

**Example:****Program**

```
import time

def func1():
    for _ in range(0,5):
        time.sleep(1)
        print("CCIT")

def func2():
    for _ in range(0,5):
        time.sleep(1)
        print("Amravati")

func1()
func2()
print("Task end")
```

**Output**

```
CCIT
CCIT
CCIT
CCIT
CCIT
Amravati
Amravati
Amravati
Amravati
Amravati
Task end
```

The below program has two functions that are linked to two child threads. These two child threads are executed parallel. The print function (task end) is in the main thread. The two child threads and main thread are not linked together, so the main thread is executed first and then the child threads are executed.

**Example:****Program**

```
import threading,time

def func1():
    for _ in range(0,5):
        time.sleep(1)
        print("CCIT")

def func2():
    for _ in range(0,5):
        time.sleep(1)
        print("Amravati")

t1=threading.Thread(target=func1)
t2=threading.Thread(target=func2)
t1.start()
t2.start()

print("Task end")
```

**Output**

```
Task end
Amravati
CCIT
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
```

To link the main thread and child thread you can use join function of thread class.

### Example:

#### Program

```
import threading,time

def func1():
    for _ in range(0,5):
        time.sleep(1)
        print("CCIT")

def func2():
    for _ in range(0,5):
        time.sleep(1)
        print("Amravati")

t1=threading.Thread(target=func1)
t2=threading.Thread(target=func2)
t1.start()
t2.start()
t1.join()
t2.join()

print("Task end")
```

#### Output

```
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
Task end
```

### Threads in Python can be created in three ways:

1. Without creating a class
2. By extending Thread class
3. Without extending Thread class

### Without creating a class

Multithreading in Python can be accomplished without creating a class as well.

**Example:****Program**

```
import threading,time

def func1():
    for _ in range(0,5):
        time.sleep(1)
        print("CCIT")

def func2():
    for _ in range(0,5):
        time.sleep(1)
        print("Amravati")

t1=threading.Thread(target=func1)
t2=threading.Thread(target=func2)
t1.start()
t2.start()
t1.join()
t2.join()

print("Task end")
```

**Output**

```
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
Task end
```

The above output shows that the first thread that is present is, the main thread. This main thread then creates a child thread that is executing the function and then the final print statement is executed again by the main thread.

## By extending the Thread class

When a child class is created by extending the Thread class, the child class represents that a new thread is executing some task. When extending the Thread class, the child class can override only two methods i.e. the `__init__()` method and the `run()` method. No other method can be overridden other than these two methods.

**Example:****Program**

```

import threading,time

class A(threading.Thread):
    def run(self):
        for _ in range(0,5):
            time.sleep(1)
            print("CCIT")

class B(threading.Thread):
    def run(self):
        for _ in range(0,5):
            time.sleep(1)
            print("Amravati")

t1=A()
t2=B()

t1.start()
t2.start()

t1.join()
t2.join()

print("Task end")

```

**Output**

```

CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
Task end

```

The above example shows that class A and B is inheriting the Thread class and the child class i.e A and B is overriding the run method. By default, the first parameter of any class function needs to be self which is the pointer to the current object. The output shows that the child thread executes the run() method and the main thread waits for the childs execution to complete. This is because of the join() function, which makes the main thread wait for the child to finish.

This method of creating threads is the most preferred method because its the standard method.

## Without Extending Thread class

To create a thread without extending the Thread class.

**Example:****Program**

```
import threading,time
class A():
    def func(self):
        for _ in range(0,5):
            time.sleep(1)
            print("CCIT")

class B():
    def func(self):
        for _ in range(0,5):
            time.sleep(1)
            print("Amravati")

a=A()
b=B()
t1=threading.Thread(target=a.func)
t2=threading.Thread(target=b.func)
t1.start()
t2.start()
t1.join()
t2.join()
print("Task end")
```

**Output**

```
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
CCIT
Amravati
Task end
```

**Example:****Program**

```
import threading,time
class A(threading.Thread):
    def run(self):
        for _ in range(0,5):
            time.sleep(1)
            print(threading.current_thread().getName())

class B(threading.Thread):
    def run(self):
        for _ in range(0,5):
            time.sleep(1)
            print(threading.current_thread().getName())

print(threading.current_thread().getName())
t1=A()
t2=B()
t1.start()
t2.start()
t1.join()
t2.join()
print(threading.current_thread().getName(),"End")
```

**Output**

```
MainThread
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
Thread-1
Thread-2
MainThread End
```

# Daemon Thread

Daemon thread is a low priority thread (in context of PVM) that runs in background to perform tasks such as garbage collection (gc) etc., they do not prevent the PVM from exiting (even if the daemon thread itself is running) when all the user threads (non-daemon threads) finish their execution. PVM terminates itself when all user threads (non-daemon threads) finish their execution, PVM does not care whether Daemon thread is running or not, if PVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself.

## Example:

### Program

```
import threading
import time
x1=0
x2=0

def task1():
    global x1
    for i in range(0,10):
        x1+=1
        print("X1:",x1)
        time.sleep(1)

def task2():
    global x2
    for i in range(0,15):
        x2+=1
        print("X2:",x2)
        time.sleep(1)

t1=threading.Thread(target=task1)
t2=threading.Thread(target=task2)
t2.setDaemon(True)
t1.start()
t2.start()
```

### Output

```
X1: 1
X2: 1
X1: 2
X2: 2
X1: 3
X2: 3
X1: 4
X2: 4
X1: 5
X2: 5
X1: 6
X2: 6
X1: 7
X2: 7
X1: 8
X2: 8
X1: 9
X2: 9
X1: 10
X2: 10
```

# Thread Synchronization

One important issue when using threads is to avoid conflicts when more than one thread needs to access a single variable or other resource. If you're not careful, overlapping accesses or modifications from multiple threads may cause all kinds of problems, and what's worse, those problems have a tendency of appearing only under heavy load, or on your production servers, or on some faster hardware that's only used by one of your customers.

There are a number of ways to avoid or solve race conditions. You won't look at all of them here, but there are a couple that are used frequently. Let's start with Lock.

To solve your race condition above, you need to find a way to allow only one thread at a time into the read-modify-write section of your code. The most common way to do this is called Lock in Python.

## Lock class

Locks are the most fundamental synchronization mechanism provided by the threading module. At any time, a lock can be held by a single thread, or by no thread at all. If a thread attempts to hold a lock that's already held by some other thread, execution of the first thread is halted until the lock is released.

Locks are typically used to synchronize access to a shared resource. For each shared resource, create a Lock object. When you need to access the resource, call acquire to hold the lock (this will wait for the lock to be released, if necessary), and call release to release it.

A Lock is an object that acts like a hall pass. Only one thread at a time can have the Lock. Any other thread that wants the Lock must wait until the owner of the Lock gives it up.

### Lock object Method

This object contains methods.

Method	Description
<b>acquire(blocking=True)</b>	Acquire a lock, blocking or non-blocking. When invoked with the blocking argument set to True (the default), block until the lock is unlocked, then set it to locked and return True.
<b>release()</b>	Release a lock. This can be called from any thread, not only the thread which has acquired the lock.
<b>locked()</b>	Return true if the lock is acquired.

**Example:****Program**

```
import threading,time
x=0
def count():
    global x
    for _ in range(0,100000):
        x += 1
t1=threading.Thread(target=count)
t2=threading.Thread(target=count)
t1.start()
t2.start()
t1.join()
t2.join()
print("Count Value",x)
```

**Output**

Count Value 139768

In above program have two child thread are that link to count function both the threads executes the count function parallel so the x variable gets overlap that why some times value of x doesn't increments.

**Example:****Program**

```
import threading,time
x=0
lock=threading.Lock()
def count(lock):
    global x
    for _ in range(0,100000):
        lock.acquire()
        x += 1
        lock.release()
t1=threading.Thread(target=count,args=(lock,))
t2=threading.Thread(target=count,args=(lock,))
t1.start()
t2.start()
t1.join()
t2.join()
print("Count Value",x)
```

**Output**

Count Value 200000

The above program use lock class so that x variable increment can be done by one thread at a time.

## Lock class using with statement

**Example:**

Program

```
import threading,time
x=0
lock=threading.Lock()

def count(lock):
    global x
    for _ in range(0,100000):
        with lock:
            x += 1

t1=threading.Thread(target=count,args=(lock,))
t2=threading.Thread(target=count,args=(lock,))
t1.start()
t2.start()
t1.join()
t2.join()
print("Count Value",x)
```

Output

Count Value 200000

## Deadlock

Lock ordering is a simple yet effective deadlock prevention mechanism. Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry.

# Multiprocessing

---

---

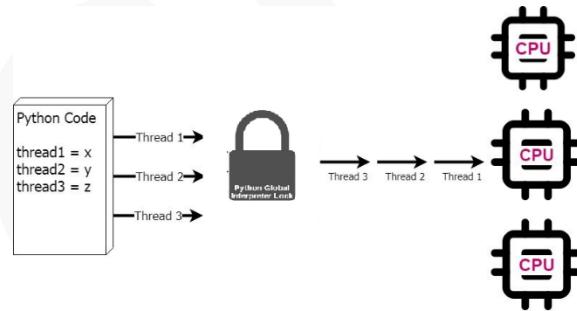


# Multiprocessing

The “multi” in multiprocessing refers to the multiple cores in a computer’s central processing unit (CPU). Computers originally had only one CPU core or processor, which is the unit that makes all our mathematical calculations possible. Today, computers typically have anywhere from 2 to 128 cores, meaning that taking advantage of more than one can dramatically improve processing time.

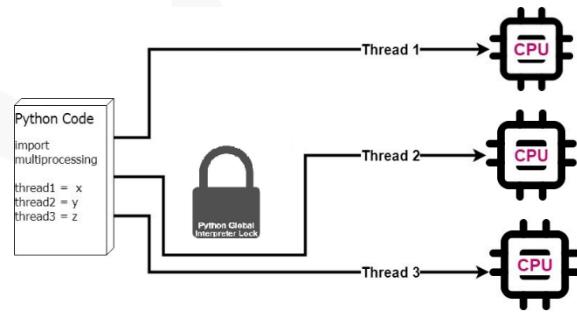
Most programs and programming languages don’t take advantage of multiple cores. These “higher-level” languages come with out-of-the-box functionality and packages that make it easier to work with data, but these languages also default to using just one core, even in machines where multiple CPUs are available. By using just that single core, these programming languages are less efficient.

In Python, single-CPU use is caused by the global interpreter lock (GIL), which allows only one thread to carry the Python interpreter at any given time. The GIL was implemented to handle a memory management issue, but as a result, Python is limited to using a single processor.



## Multiprocessing can dramatically improve processing speed

Bypassing the GIL when executing Python code allows the code to run faster because we can now take advantage of multiprocessing. Python’s built-in multiprocessing module allows us to designate certain sections of code to bypass the GIL and send the code to multiple processors for simultaneous execution.



In this simplified example, assuming all three threads had identical runtimes, the multiprocessing solution would cut total execution time by a third. But this reduction isn’t exactly proportionate to the number of processors available because of the overhead involved in creating multiprocessing processes, but the gains represent a significant improvement over single-core operations.

# multiprocessing module

Python introduced multiprocessing module to let us write parallel code. The multiprocessing Python module contains two classes capable of handling tasks. The Process class sends each task to a different processor, and the Pool class sends sets of tasks to different processors.

Method	Description
<b>active_children()</b>	Return list of all live children of the current process.
<b>cpu_count()</b>	Return the number of CPUs in the system.
<b>current_process()</b>	Return the Process object corresponding to the current process.
<b>parent_process()</b>	Return the Process object corresponding to the parent process of the current_process().

## Process class

Python multiprocessing Process class is an abstraction that sets up another Python process, provides it to run code and a way for the parent application to control execution. Process objects represent activity that is run in a separate process. The Process class has equivalents of all the methods of threading.Thread.

### Syntax:

```
Process(group=None, target=None, name=None, args=(), kwargs={}, daemon=None)
```

**group** : group should always be None; it exists solely for compatibility with threading.Thread.

**name** : name is the process name.

**target** : target is the callable object to be invoked by the run() method. It defaults to None

**args** : Argument to be passed to target function.

**kwargs** : Keyword argument dictionary for the target invocation. Defaults to {}.

Note: If a subclass overrides the constructor, it must make sure it invokes the base class constructor (Process.\_\_init\_\_()) before doing anything else to the process.

## Process object Attribute

This object contains attributes.

Attribute	Description
<b>Daemon</b>	The process' s daemon flag, a Boolean value. This must be set before start() is called.
<b>Name</b>	The process' s name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.
<b>Pid</b>	Return the process ID.

## Process object Method

This object contains method.

Method	Description
<b>start()</b>	Start the process' s activity. This must be called at most once per process object. It arranges for the object' s run() method to be invoked in a separate process
<b>run()</b>	Method representing the process' s activity. You may override this method in a subclass. The standard run() method invokes the callable object passed to the objects constructor as the target argument
<b>join(timeout=None)</b>	If the optional argument timeout is None (the default), the method blocks until the process whose join() method is called terminates. If timeout is a positive number, it blocks at most timeout seconds.
<b>terminate()</b>	Terminate the process.
<b>kill()</b>	Same as terminate()
<b>close()</b>	Close the Process object
<b>is_alive()</b>	Return whether the process is alive.

**Example:****Program**

```

import time
import multiprocessing

x1=0
def func():
    global x1
    for i in range(5):
        x1+=1
        print(x1)
        time.sleep(1)

if __name__=='__main__':
    p1=multiprocessing.Process(target=func)
    p2=multiprocessing.Process(target=func)
    p1.start()
    p2.start()

```

**Output**

```

1
1
2
2
3
3
4
4
5
5

```

**Example:****Program**

```

import multiprocessing
import time

class task_A(multiprocessing.Process):
    def run(self):
        for i in range(1,5):
            time.sleep(1)
            print(i)

if __name__=='__main__':
    a=task_A()
    a.start()
    b=task_A()
    b.start()

```

**Output**

```

1
1
2
2
3
3
4
4
5
5

```

# Daemon Process

A daemon process is a background process that is not under the direct control of the user. This process is usually started when the system is bootstrapped and it terminates with the system shut down.

Usually the parent process of the daemon process is the init process. This is because the init process usually adopts the daemon process after the parent process forks the daemon process and terminates.

Python multiprocessing module allows us to have daemon processes through its `daemonic` option. Daemon processes or the processes that are running in the background follow similar concept as the daemon threads. To execute the process in the background, we need to set the `daemonic` flag to true. The daemon process will continue to run as long as the main process is executing and it will terminate after finishing its execution or when the main program would be killed.

## Example:

### Program

```
import time
import multiprocessing

x1=0

def task_A():
    global x1
    for i in range(10):
        x1+=1
        print(x1)
        time.sleep(0.5)

def task_B():
    global x1
    for i in range(15):
        x1+=1
        print(x1)
        time.sleep(0.5)

if __name__=='__main__':
    p1=multiprocessing.Process(target=task_A)
    p2=multiprocessing.Process(target=task_B)
    p2.daemon=True
    p1.start()
    p2.start()
    p1.join()
```

### Output

```
x2: 1
x1: 1
x2: 2
x1: 2
x2: 3
x1: 3
x2: 4
x1: 4
x2: 5
x1: 5
x2: 6
x1: 6
x2: 7
x1: 7
x2: 8
x1: 8
x2: 9
x1: 9
x2: 10
x1: 10
x2: 11
```

# Process Synchronization

Process Synchronization means sharing system resources by processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

multiprocessing module provides a Lock class to deal with the race conditions.

## Lock class

Locks are the most fundamental synchronization mechanism provided by the multiprocessing module. At any time, a lock can be held by a single process, or by no process at all. If a process attempts to hold a lock that's already held by some other process, execution of the first process is halted until the lock is released.

Locks are typically used to synchronize access to a shared resource. For each shared resource, create a Lock object. When you need to access the resource, call acquire to hold the lock (this will wait for the lock to be released, if necessary), and call release to release it

A Lock is an object that acts like a hall pass. Only one thread at a time can have the Lock. Any other process that wants the Lock must wait until the owner of the Lock gives it up.

### Lock object Method

This object contains methods.

Method	Description
<b>acquire(blocking=True)</b>	Acquire a lock, blocking or non-blocking. When invoked with the blocking argument set to True (the default), block until the lock is unlocked, then set it to locked and return True.
<b>release()</b>	Release a lock. This can be called from any thread, not only the thread which has acquired the lock.
<b>locked()</b>	Return true if the lock is acquired.

# multiprocessing shared memory

This module provides a class, SharedMemory, for the allocation and management of shared memory to be accessed by one or more processes on a multicore or symmetric multiprocessor (SMP) machine.

To use shared memory to hold a single primitive value in common among two or more processes, multiprocessing supplies the class Value; for a fixed-length array of primitive values, the class Array. For more flexibility (including nonprimitive values, and “sharing” among different systems joined by a network but sharing no memory) at the cost of higher overhead, multiprocessing supplies the class Manager, which is a subclass of Process.

It is possible to create shared objects using shared memory which can be inherited by child processes.

## class Value

The object of this type use to share an single value between the processes.

Return a ctypes object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the value attribute of a Value.

### Syntax:

```
Value(typecode_or_type, *args, lock=True)
```

typecode\_or\_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module.

\*args is passed on to the constructor for the type.

Lock If lock is True (the default) then a new recursive lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value.

Type code	C Type	Python Type
'b'	signed char	int
'B'	unsigned char	int
'i'	signed int	int
'I'	unsigned int	int
'l'	signed long	int
'L'	unsigned long	int
'q'	signed long long	int
'Q'	unsigned long long	int
'f'	float	float
'd'	double	float

## value object attributes and methods

An instance of the class Value supplies the following attributes and methods

Method	Description
<b>Value</b>	A read/write attribute, used to set and get v' s underlying primitive value.
<b>get_lock()</b>	Returns (but neither acquires nor releases) the lock guarding v

### Example:

#### Program

```
import multiprocessing
import time

def task_A(data,lock):
    for i in range(0,100000):
        lock.acquire()
        data.value=data.value+1
        lock.release()

if __name__=='__main__':
    lock=multiprocessing.Lock()
    data=multiprocessing.Value('i')
    p1=multiprocessing.Process(target=task_A,args=(data,lock))
    p2=multiprocessing.Process(target=task_A,args=(data,lock))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print(data.value)
```

#### Output

200000

# class Array

The object of this type use to share an bytestring between the processes.

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

## Syntax:

```
Array(typecode_or_type, size_or_initializer, *args, lock=True)
```

typecode\_or\_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module.

size\_or\_initializer can be an iterable, used to initialize the array; alternatively, it can be an integer, used as the length of the array (in this case, each item of the array is initialized to 0).

\*args is passed on to the constructor for the type.

Lock If lock is True (the default) then a new recursive lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value.

## Example:

### Program

```
import multiprocessing

def printer(data,lock):
    lock.acquire()
    print(data.value)
    lock.release()

if __name__=='__main__':
    lock=multiprocessing.Lock()
    data=multiprocessing.Array('c',b"Python @ CCIT")
    p1=multiprocessing.Process(target=printer,args=(data,lock))
    p2=multiprocessing.Process(target=printer,args=(data,lock))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

### Output

```
b"Python @ CCIT"
b"Python @ CCIT"
```

# Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages shared objects. Other processes can access the shared objects by using proxies.

## Syntax:

```
multiprocessing.Manager()
```

Returns a started SyncManager object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

## SyncManager

A subclass of BaseManager which can be used for the synchronization of processes. Objects of this type are returned by `multiprocessing.Manager()`.

Its methods create and return Proxy Objects for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

## object attributes and methods

An instance of the class supplies the following attributes and methods

Method	Description
<b>Array(typecode, sequence)</b>	Create an array and return a proxy for it.
<b>Value(typecode, value)</b>	Create an object with a writable value attribute and return a proxy for it.
<b>dict(sequence)</b>	Create a shared dict object and return a proxy for it.
<b>list(sequence)</b>	Create a shared list object and return a proxy for it.

**Example:****Program**

```
import multiprocessing

def task_A(data,lock):
    for i in range(0,len(data)-1):
        lock.acquire()
        if data[i]>data[i+1]:
            data[i],data[i+1]=data[i+1],data[i]
        lock.release()

if __name__=='__main__':
    lock=multiprocessing.Lock()
    mgr=multiprocessing.Manager()
    data=mgr.list([421,571,352,415,115,487,268,984,266,687,357])
    for i in range(0,len(data)-1):
        p=multiprocessing.Process(target=task_A,args=(data,lock))
        p.start()
        p.join()
    print("Sorted list...")
    print(data)
```

**Output**

```
Sorted list...
[115, 266, 268, 352, 357, 415, 421, 487, 571, 687, 984]
```

# BaseManager

Create a BaseManager object.

Once created one should call start() or get\_server().serve\_forever() to ensure that the manager object refers to a started manager process.

## Syntax:

```
multiprocessing.BaseManager([address[, authkey]])
```

address is the address on which the manager process listens for new connections. If address is None then an arbitrary one is chosen.

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If authkey is None then current\_process().authkey is used. Otherwise authkey is used and it must be a byte string.

## object attributes and methods

An instance of the class supplies the following attributes and methods

Method	Description
<b>start()</b>	Start a subprocess to start the manager.
<b>register(typeid, callable)</b>	A classmethod which can be used for registering a type or callable with the manager class.  typeid is a “type identifier” which is used to identify a particular type of shared object. This must be a string.  callable is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the connect() method, or if the create_method argument is False then this can be left as None.
<b>connect()</b>	Connect a local manager object to a remote manager process
<b>shutdown()</b>	Stop the process used by the manager. This is only available if start() has been used to start the server process.

## Example:

### Program

```
import multiprocessing
import multiprocessing.managers

class MathsClass:
    def __init__(self):
        self.x=0
    def count(self):
        self.x=self.x+1
        return self.x
    def display(self):
        return self.x

def printer(obj,lock):
    for i in range(0,100000):
        lock.acquire()
        obj.count()
        lock.release()

if __name__=='__main__':
    lock=multiprocessing.Lock()
    mgr=multiprocessing.managers.BaseManager()
    mgr.register('math',MathsClass)
    mgr.start()
    mth=mgr.math()
    p1=multiprocessing.Process(target=printer,args=(mth,lock))
    p2=multiprocessing.Process(target=printer,args=(mth,lock))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print(mth.display())
```

### Output

```
200000
```

# Process Pools

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

## Syntax:

```
multiprocessing.pool.Pool(processes, initializer, initargs, maxtasksperchild, context)
```

processes is the number of worker processes to use. If processes is None then the number returned by os.cpu\_count() is used.

If initializer is not None then each worker process will call initializer(\*initargs) when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default maxtasksperchild is None, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function multiprocessing.Pool() or the Pool() method of a context object. In both cases context is set appropriately.

## object attributes and methods

An instance of the class supplies the following attributes and methods

Method	Description
<b>apply(func, args, kwds)</b>	Call func with arguments args and keyword arguments kwds. It blocks until the result is ready.
<b>map(func, iterable)</b>	A parallel equivalent of the map() built-in function (it supports only one iterable argument though, for multiple iterables see starmap()). It blocks until the result is ready.  This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks.
<b>starmap(func, iterable)</b>	Like map() except that the elements of the iterable are expected to be iterables that are unpacked as arguments.  Hence an iterable of [(1,2), (3, 4)] results in [func(1,2), func(3,4)].

<b>close()</b>	Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.
<b>terminate()</b>	Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected terminate() will be called immediately.
<b>join()</b>	Wait for the worker processes to exit. One must call close() or terminate() before using join().

**Example:****Program**

```
import multiprocessing

def sqr(x):
    return x*x

if __name__=='__main__':
    lst=[2,3,4,5,6,7,8]
    pool=multiprocessing.Pool()
    result=pool.map(sqr,lst)
    print(result)
```

**Output**

[4, 9, 16, 25, 36, 49, 64]

**Example:****Program**

```
import multiprocessing

def sqr(x,y):
    return x+y

if __name__=='__main__':
    lst=[(0,1),(1,1),(2,1),(3,2),(5,3),(8,5)]
    pool=multiprocessing.Pool()
    result=pool.starmap(sqr,lst)
    print(result)
```

**Output**

[1, 2, 3, 5, 8, 13]

# subprocess

---

---

# subprocess Module

A running program is called a process. Each process has its own system state, which includes memory, lists of open files, a program counter that keeps track of the instruction being executed, and a call stack used to hold the local variables of functions. Normally, a process executes statements one after the other in a single sequence of control flow, which is sometimes called the main thread of the process. At any given time, the program is only doing one thing.

A program can create new processes using library functions such as those found in the subprocess modules such as subprocess.Popen(), etc. However, these processes, known as subprocesses, run as completely independent entities-each with their own private system state and main thread of execution. Because a subprocess is independent, it executes concurrently with the original process. That is, the process that created the subprocess can go on to work on other things while the subprocess carries out its own work behind the scenes. The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

## Popen Class

Python subprocess Popen is used to execute a child program in a new process. We can use it to run some shell commands. The underlying process creation and management in this module is handled by the Popen class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions. subprocess.Popen() executes a child program in a new process.

### Syntax:

```
Popen(args, stdin=None, stdout=None, stderr=None, shell=False, cwd=None)
```

args should be a string, or a sequence of program arguments. The program to execute is normally the first item in the args sequence or string, but can be explicitly set by using the executable argument.

If shell is true, the specified command will be executed through the shell.

stdin, stdout and stderr specify the executed programs' standard input, standard output and standard error file handles, respectively. Valid values are PIPE, DEVNULL, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. DEVNULL indicates that the special file os.devnull will be used. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent.

If cwd is not None, the current directory will be changed to cwd before the child is executed.

## object Attribute

This object contains attributes.

Attribute	Description
<b>args</b>	The args argument as it was passed to Popen – a sequence of program arguments or else a single string.
<b>stdin</b>	If the stdin argument is PIPE, this attribute is a file object that provides input to the child process. Otherwise, it is None.
<b>stdout</b>	If the stdout argument is PIPE, this attribute is a file object that provides output from the child process. Otherwise, it is None.
<b>stderr</b>	If the stderr argument is PIPE, this attribute is file object that provides error output from the child process. Otherwise, it is None.
<b>pid</b>	The process ID of the child process.
<b>returncode</b>	The child return code. A None value indicates that the process hasn't terminated yet. A negative value -N indicates that the child was terminated by signal N (UNIX only).

## object methods

An instance of the class supplies the following methods

Method	Description
<b>poll()</b>	Check if child process has terminated. Set and return returncode attribute. Otherwise, returns None.
<b>wait(timeout=None)</b>	Wait for child process to terminate. Set and return returncode attribute. If the process does not terminate after timeout seconds, raise a TimeoutExpired exception. It is safe to catch this exception and retry the wait.
<b>communicate(input=None, timeout=None)</b>	Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the returncode attribute. The optional input argument should be data to be sent to the child process, or None, if no data should be sent to the child.
<b>terminate()</b>	Stop the child.

**Example:**

## Program

```
import subprocess

out=subprocess.Popen("echo 'Welcome to CCIT' ",stdout=subprocess.PIPE,
shell=True)
print(out.communicate())
```

## Output

```
(b"'Welcome to CCIT' \r\n", None)
```

**Example:**

## Program

```
import subprocess

out=subprocess.Popen('dir',stdout=subprocess.PIPE,shell=True,cwd="c://temp")
print(out.communicate()[0].decode())
```

## Output

```
Volume in drive C has no label.
Volume Serial Number is F442-B653
Directory of c:\temp
09-Jul-20  03:07 PM    <DIR>.
09-Jul-20  03:07 PM    <DIR>..
09-Jul-20  03:07 PM            0 demo.py
18-Jul-19  10:14 AM            40 num.txt
09-Jul-20  03:06 PM    <DIR>sub folder
06-Jul-20  02:02 AM            90text.txt
                           3 File(s)           130 bytes
                           3 Dir(s)  20,079,075,328 bytes free
```

**Example:**

## Program

```
circle.py
-----
def circle(r):
    A=3.14*r*r
    C=2*3.14*r
    print("Area is",A)
    print("Circumference is",C)
circle(5)
```

## Output

```
Area is 78.5
```

```
demo.py
-----
import subprocess

out=subprocess.Popen("python
circle.py",stdout=subprocess.PIPE,shell=True,cwd="c://temp/")
print(out.communicate()[0].decode())
```

```
Circumference is 31.400000000000002
```

## subprocess function

The recommended approach to invoking subprocesses is to use the run() function for all use cases it can handle. For more advanced use cases, the underlying Popen interface can be used directly.

- **subprocess.run(args, \*, stdin=None, input=None, stdout=None, stderr=None, capture\_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text = None)**

Run the command described by args. Wait for command to complete, then return a CompletedProcess instance.

- **subprocess.call(args, \*, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None)**

This function is used to run a command and get the return code of the command. Run the command described by args. Wait for command to complete, then return the returncode attribute.

- **subprocess.check\_call(args, \*, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None)**

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise CalledProcessError.

- **subprocess.check\_output(args, \*, stdin=None, stderr=None, shell=False, cwd=None, encoding=None, errors=None, timeout=None)**

Run command with arguments and return its output. If the return code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute and any output in the output attribute.

args: should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in args if args is a sequence. If args is a string, the interpretation is platform-dependent. It is recommended to pass args as a sequence.

shell: shell argument (which defaults to False) specifies whether to use the shell as the program to execute. If shell is True, it is recommended to pass args as a string rather than as a sequence.

On Unix with shell=True, the shell defaults to /bin/sh.

stdin, stdout and stderr: specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are PIPE, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, stderr

can be STDOUT, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

cwd: is not None the child's current directory will be changed to cwd before it is executed. Note that this directory is not considered when searching the executable, so we can't specify the program's path relative to cwd.

If capture\_output is true, stdout and stderr will be captured.

If the timeout expires, the child process will be killed and waited for. The TimeoutExpired exception will be re-raised after the child process has terminated.

If check is true, and the process exits with a non-zero exit code, a CalledProcessError exception will be raised.

## Class CompletedProcess

The return value from run(), representing a process that has finished.

Attribute	Description
<b>args</b>	The arguments used to launch the process. This may be a list or a string.
<b>returncode</b>	Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully
<b>stdout</b>	Captured stdout from the child process. A bytes sequence, or a string if run() was called with an encoding, errors, or text=True. None if stdout was not captured.
<b>stderr</b>	Captured stderr from the child process. A bytes sequence, or a string if run() was called with an encoding, errors, or text=True. None if stderr was not captured.

### Example:

#### Program

```
import subprocess

out=subprocess.run("echo 'Welcome to CCIT' ",shell=True, capture_output=True)
print(out)
```

#### Output

```
CompletedProcess(args="echo 'Welcome to CCIT' ", returncode=0,
stdout=b"Welcome to CCIT' \r\n",
stderr=b'')
```

**Example:****Program**

```
import subprocess

out=subprocess.run("dir",shell=True,capture_output=True cwd="c://temp")
print(out.stdout.decode())

out=subprocess.call("md sub_folder2",shell=True,cwd="c://temp")

out=subprocess.run("dir",shell=True,capture_output=True cwd="c://temp")
print(out.stdout.decode())
```

**Output**

```
Volume in drive C has no label.
Volume Serial Number is F442-B653
Directory of c:\temp

10-Jul-20  02:40 AM    <DIR>      .
10-Jul-20  02:40 AM    <DIR>      ..
09-Jul-20  10:43 PM          102 circle.py
18-Jul-19  10:14 AM          40 num.txt
09-Jul-20  03:06 PM    <DIR>      sub folder
06-Jul-20  02:02 AM          90 text.txt
                           3 File(s)       232 bytes
                           3 Dir(s)  19,686,195,200 bytes free
```

```
Volume in drive C has no label.
Volume Serial Number is F442-B653
Directory of c:\temp

10-Jul-20  02:42 AM    <DIR>      .
10-Jul-20  02:42 AM    <DIR>      ..
09-Jul-20  10:43 PM          102 circle.py
18-Jul-19  10:14 AM          40 num.txt
09-Jul-20  03:06 PM    <DIR>      sub folder
10-Jul-20  02:42 AM    <DIR>      sub_folder2
06-Jul-20  02:02 AM          90 text.txt
                           3 File(s)       232 bytes
                           4 Dir(s)  19,689,652,224 bytes free
```

# Tkinter

---

---

# Tkinter

The Tk widget library originates from the Tool Command Language (Tcl) programming language. Tcl and Tk were created by John Ousterman while he was a professor at Berkeley in the late 1980s as an easier way to program engineering tools being used at the university. Because of its speed and relative simplicity, Tcl/Tk rapidly grew in popularity among academic, engineering, and Unix programmers. Much like Python itself, Tcl/Tk originated on the Unix platform and only later migrated to macOS and Windows. Tk's practical intent and Unix roots still inform its design today, and its simplicity compared to other toolkits is still a major strength.

Tkinter is a Python interface to the Tk GUI library and has been a part of the Python standard library since 1994 with the release of Python version 1.1, making it the de facto GUI library for Python.

Tkinter is also very quick and easy to learn. Code can be written both procedurally or using object-oriented practices (which is the preferred style for anything nonexperimental), and runs perfectly on any operating system supporting Python development, including Windows, macOS, and Linux.

Python coders who want to build a GUI have several toolkit options to choose from; unfortunately, Tkinter is often maligned or ignored as a legacy option. To be fair, it's not a glamorous technology that you can describe in trendy buzzwords and glowing hype. However, Tkinter is not only adequate for a wide variety of applications, it also has the following advantages.

- **It's in the standard library:** With few exceptions, Tkinter is available wherever Python is available. There is no need to install pip, create virtual environments, compile binaries, or search the web for installation packages. For simple projects that need to be done quickly, this is a clear advantage.
- **It's stable:** While Tkinter development has not stopped, it is slow and evolutionary. The API has been stable for years, the changes mainly being additional functionality and bug fixes. Your Tkinter code will likely run unaltered for years or decades to come.
- **It's only a GUI toolkit:** Unlike some other GUI libraries, Tkinter doesn't have its own threading library, network stack, or filesystem API. It relies on regular Python libraries for such things, so it's perfect for applying a GUI to existing Python code.
- **It's simple and no-nonsense:** Tkinter is straightforward, old-school object-oriented GUI design. To use Tkinter, you don't have to learn hundreds of widget classes, a markup or templating language, a new programming paradigm, client-server technologies, or a different programming language

# Tkinter Modules

tkinter is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named \_tkinter. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, tkinter includes a number of Python modules, tkinter.constants being one of the most important. Importing tkinter will automatically import tkinter.constants, so, usually, to use Tkinter all you need is a simple import statement:

## Syntax:

```
import tkinter
```

# Tk class

The Tk class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter

## Syntax:

```
Tk(screenName=None, baseName=None, className='Tk', useTk=1)
```

Return a new Toplevel widget on screen SCREENNAME. A new Tcl interpreter will be created. BASENAME will be used for the identification of the profile file (see readprofile).

## object methods

An instance of the class supplies the following methods

Method	Description
<b>title(string)</b>	Set the title of this window.
<b>maxsize(width, height)</b>	Set max WIDTH and HEIGHT for this widget.
<b>minsize(width, height)</b>	Set min WIDTH and HEIGHT for this widget.

<b>resizable(width, height)</b>	whether this window can be resized in WIDTH or HEIGHT. Both values are Boolean values.
<b>geometry("WidthxHeight")</b>	Sets the size of window.
<b>destroy( )</b>	Closes the window.
<b>configure(**options)</b>	Sets the attributes of window.

## configure methods

Attribute	Description
<b>background , bg</b>	background color
<b>borderwidth , bd</b>	Size of border default is 2.
<b>height</b>	Sets Height
<b>width</b>	Sets width
<b>font</b>	font for window
<b>padx</b>	Sets padding x-axis
<b>pady</b>	Sets padding y-axis
<b>cursor</b>	Cursor for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>relief</b>	window style Values: FLAT , RAISED , SUNKEN , GROOVE , RIDGE etc.
<b>menu</b>	It sets menu for window
<b>highlightcolor</b>	It sets the highlightcolor of window
<b>highlightbackground</b>	It sets the highlightcolorbackground of window
<b>highlightthickness</b>	It sets the highlightthickness width

# mainloop

The method mainloop has an important role for TkInter, it is waiting for events and updating the GUI. But this method is blocking the code after it. You have a conflict, if the core of your application has also a blocking loop that is waiting for some events.

- This is the last statement of our GUI application.
- When the main loop is entered, the GUI takes over control from there.
- All other actions are handled via callbacks i.e event handlers.
- This method is of class Tk.

## Syntax:

```
mainloop()
```

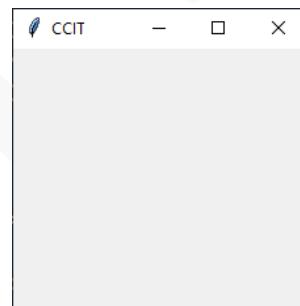
## Example:

### Program

```
import tkinter

win=tkinter.Tk()
win.title('CCIT')
win.mainloop()
```

### Output



## Example:

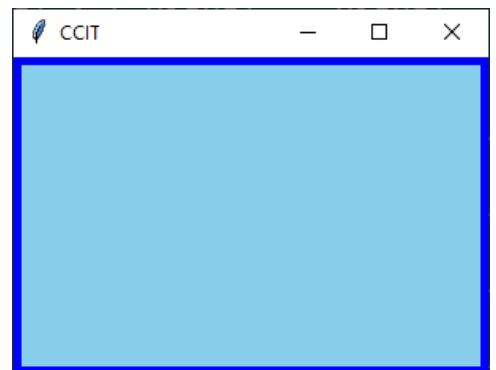
### Program

```
import tkinter

class Win(tkinter.Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.geometry("300x200")
        self.configure(bg="skyblue",highlightcolor='blue',highlightthickness=5)

win=Win()
win.mainloop()
```

### Output



# Button

The Button widget is a standard Tkinter widget, which is used for various kinds of buttons. A button is a widget which is designed for the user to interact with, i.e. if the button is pressed by mouse click some action might be started. They can also contain text and images like labels. While labels can display text in various fonts, a button can only display text in a single font. The text of a button can span more than one line.

## Syntax:

```
Button(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>pack( )</b>	To add button into container(window).
<b>place( )</b>	To add button into container(window).
<b>grid( )</b>	To add button into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure methods

Attribute	Description
<b>activebackground</b>	activebackground color
<b>activeforeground</b>	Activeforeground color
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>command</b>	Function or method to be called when the button is clicked.
<b>height</b>	Sets Height
<b>width</b>	Sets width
<b>font</b>	Text font to be used for the button's label.

<b>padx</b>	Sets padding x-axis
<b>pady</b>	Sets padding y-axis
<b>image</b>	Image to be displayed on the button (instead of text).
<b>underline</b>	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>state</b>	Set this option to DISABLED to make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>relief</b>	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, flat ,and RIDGE.
<b>highlightcolor</b>	It sets the highlightcolor of window
<b>highlightbackground</b>	It sets the highlightcolorbackground of window
<b>highlightthickness</b>	It sets the highlightthickness width

### Example:

#### Program

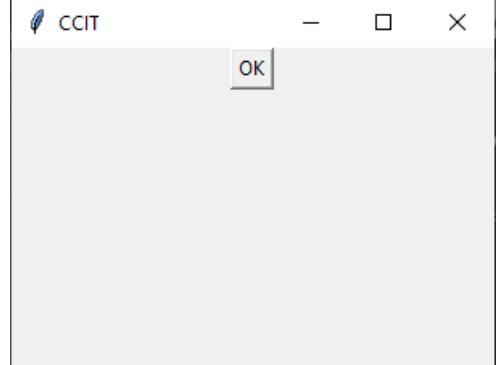
```
import tkinter

class Win(tkinter.Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.geometry("300x200")
        self.btn=tkinter.Button(self,text="OK")
        self.btn.configure(command=self.message)
        self.btn.pack()

    def message(self):
        print("Hello World")

win=Win()
win.mainloop()
```

#### Output



**Example:****Program**

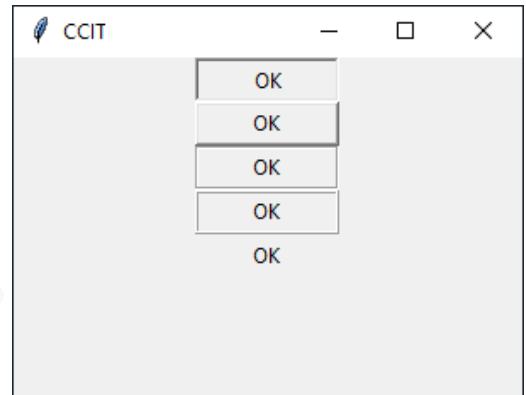
```

import tkinter

class Win(tkinter.Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.geometry("300x200")
        self.btn1=tkinter.Button(self,text="OK")
        self.btn1.configure(padx=30,relief='sunken')
        self.btn1.pack()
        self.btn2=tkinter.Button(self,text="OK")
        self.btn2.configure(padx=30,relief='raised')
        self.btn2.pack()
        self.btn3=tkinter.Button(self,text="OK")
        self.btn3.configure(padx=30,relief='groove')
        self.btn3.pack()
        self.btn4=tkinter.Button(self,text="OK")
        self.btn4.configure(padx=30,relief='ridge')
        self.btn4.pack()
        self.btn=tkinter.Button(self,text="OK")
        self.btn.configure(padx=30,relief='flat')
        self.btn.pack()

win=Win()
win.mainloop()

```

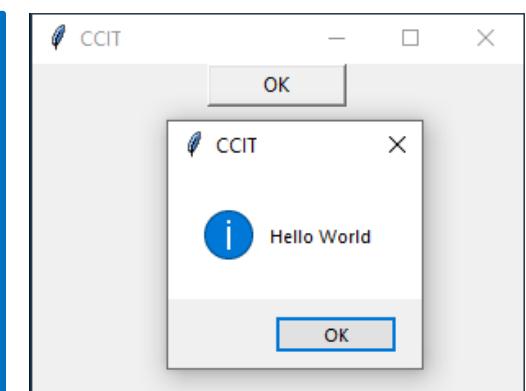
**Output****Example:****Program**

```

import tkinter
from tkinter.messagebox import *
class Win(tkinter.Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.geometry("300x200")
        self.btn=tkinter.Button(self,text="OK")
        self.btn.configure(padx=30,command=
self.message )
        self.btn.pack()
    def message(self):
        showinfo(title="CCIT",message="Hello World")

win=Win()
win.mainloop()

```

**Output**

# Entry

Entry widgets are the basic widgets of Tkinter used to get input, i.e. text strings, from the user of an application. This widget allows the user to enter a single line of text. If the user enters a string, which is longer than the available display space of the widget, the content will be scrolled. This means that the string cannot be seen in its entirety. The arrow keys can be used to move to the invisible parts of the string. If you want to enter multiple lines of text, you have to use the text widget. An entry widget is also limited to single font.

## Syntax:

```
Entry(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>get()</b>	Gets the current contents of the entry field. It will return the string.
<b>insert( index, data )</b>	Inserts data at specified index.
<b>delete(first,last=None)</b>	Deletes chars from first to last. If the second argument is omitted, only the single character at position first is deleted. If last=END is set then it will remove all chars.
<b>select_range(start,end)</b>	Selects text in specified range. It copies selected text into clipboard.
<b>select_clear()</b>	Clears the selection.
<b>select_present( )</b>	If there is a selection, returns true, else returns false.
<b>select_to(index)</b>	Selects all the text from the ANCHOR position up to but not including the character at the given index.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure methods

<b>Attribute</b>	<b>Description</b>
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>show</b>	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
<b>font</b>	Text font to be used for the button's label.
<b>textvariable</b>	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>state</b>	Set this option to DISABLED to make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>relief</b>	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, flat ,and RIDGE.
<b>selectbackground</b>	The background color to use displaying selected text.
<b>selectborderwidth</b>	The width of the border to use around selected text. The default is one pixel.
<b>selectforeground</b>	The foreground (text) color of selected text.
<b>highlightcolor</b>	It sets the highlightcolor of window
<b>highlightbackground</b>	It sets the highlightcolorbackground of window
<b>highlightthickness</b>	It sets the highlightthickness width

**Example:****Program**

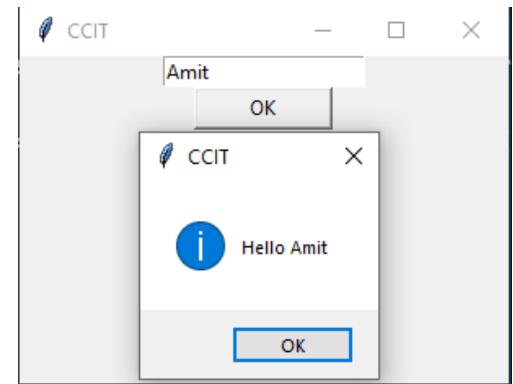
```

import tkinter
from tkinter.messagebox import *
class Win(tkinter.Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.geometry("300x200")
        self.ety=tkinter.Entry(self)
        self.ety.pack()
        self.btn=tkinter.Button(self,text="OK")
        self.btn.configure(padx=30,command=self.show)
        self.btn.pack()

    def show(self):
        data=self.ety.get()
        showinfo(title="CCIT",message="Hello "+data)

win=Win()
win.mainloop()

```

**Output****Example:****Program**

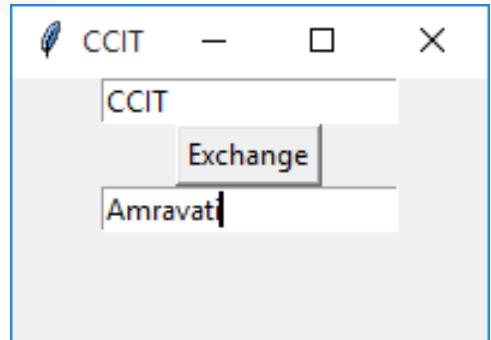
```

from tkinter import *
class MyWin(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.t1=Entry(self)
        self.t1.pack()
        self.btn=Button(self,text="Exchange",command=
self.exg)
        self.btn.pack()
        self.t2=Entry(self)
        self.t2.pack()

    def exg(self):
        s1=self.t1.get()
        s2=self.t2.get()
        self.t1.delete(0,last=END)
        self.t2.delete(0,last=END)
        self.t2.insert(0,s1)
        self.t1.insert(0,s2)

win=MyWin()
win.mainloop()

```

**Output**

# Variable Classes

Some widgets (like text entry widgets, radio buttons and so on) can be connected directly to application variables by using special options: variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value. These Tkinter control variables are used like regular Python variables to keep certain values. It's not possible to hand over a regular Python variable to a widget through a variable or textvariable option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in the Tkinter module.

Variable classes are

- BooleanVar
- DoubleVar
- IntVar
- StringVar

## Syntax:

```
var-name=var_class()
```

## For ex:

- x = StringVar() Holds a string; default value ""
- x = IntVar() Holds an integer; default value 0
- x = DoubleVar() Holds a float; default value 0.0
- x = BooleanVar() Holds a boolean, returns 0 for False and 1 for True

## object methods

An instance of the class supplies the following methods

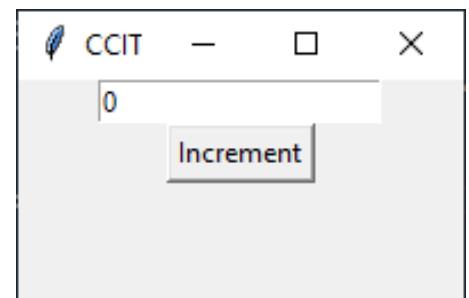
Method	Description
<b>get()</b>	The get method returns the current value of the variable.
<b>set(value)</b>	The set method updates the variable.

**Example:****Program**

```
from tkinter import *

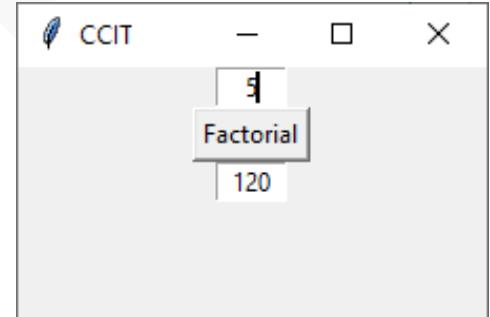
class MyWin(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.a=IntVar()
        self.t1=Entry(self,textvariable=self.a)
        self.t1.pack()
        self.btn=Button(self,text="Increment",command=
self.incr)
        self.btn.pack()
    def incr(self):
        n=self.a.get()
        n=n+1
        self.a.set(n)

win=MyWin()
win.mainloop()
```

**Output****Example:****Program**

```
import tkinter as tk
from tkinter import *

win=tk.Tk()
win.title("CCIT")
def cal():
    n=a.get()
    f=1
    for i in range(1,n+1):
        f=f*i
    b.set(f)
a=IntVar()
a.set(0)
b=IntVar()
b.set(0)
t1=Entry(win,textvariable=a,width=5,justify=CENTER)
t1.pack()
btn=Button(win,text="Factorial",command=cal)
btn.pack()
t2=Entry(win,textvariable=b,width=5,justify=CENTER)
t2.pack()
win.mainloop()
```

**Output**

# Label

This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want. It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines.

## Syntax:

```
Label(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure methods

Attribute	Description
<b>text</b>	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\\n") will force a line break.
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>image</b>	To display a static image in the label widget, set this option to an image object.
<b>bitmap</b>	Set this option equal to a bitmap or image object and the label will display that graphic.

<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>underline</b>	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>font</b>	Text font to be used for the button's label.
<b>padx</b>	Sets padding x-axis
<b>pady</b>	Sets padding y-axis
<b>textvariable</b>	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.

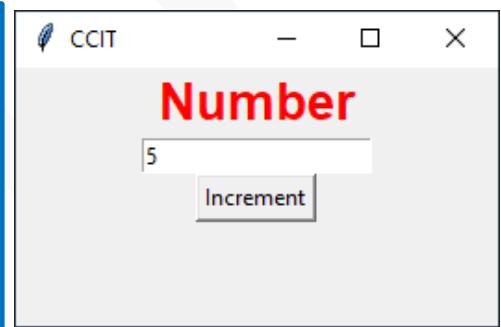
**Example:****Program**

```
from tkinter import *

class MyWin(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.title('CCIT')
        self.a=IntVar()
        self.lbl=Label(self,text="Number")
        self.lbl.configure(fg="red", font="TimesRoman
20 bold")
        self.lbl.pack()
        self.t1=Entry(self,textvariable=self.a)
        self.t1.pack()
        self.btn=Button(self,text="Increment",command
=self.incr)
        self.btn.pack()

    def incr(self):
        n=self.a.get()
        n=n+1
        self.a.set(n)

win=MyWin()
win.mainloop()
```

**Output**

# Layout management

Layout Managers arranges widgets on the screen. It decides the size and position of components. Widgets can provide hint about size and alignment information to geometry managers. But final decision is taken by Layout (geometry) managers on the positioning and sizing.

When we design the GUI of our application, we decide what widgets we will use and how we will organize those widgets in the application. To organize our widgets, we use specialized non-visible objects called layout managers. There are two kinds of widgets: containers and their children. The containers group their children into suitable layouts.

Tkinter has three built-in layout managers: the pack, grid, and place managers. The place geometry manager positions widgets using absolute positioning. The pack geometry manager organizes widgets in horizontal and vertical boxes. The grid geometry manager places widgets in a two dimensional grid.

Tkinter provides three layout managers:

1. pack
2. grid
3. place

## Pack managers

Pack is the easiest to use of the three geometry managers of Tk and Tkinter. Instead of having to declare precisely where a widget should appear on the display screen, we can declare the positions of widgets with the pack command relative to each other. The pack command takes care of the details. Though the pack command is easier to use, this layout managers is limited in its possibilities compared to the grid and place managers. For simple applications it is definitely the manager of choice.

### Syntax:

```
widget.pack( pack_options )
```

### side

- Determines which side of the parent widget packs against:
- TOP (default), BOTTOM, LEFT, or RIGHT.

**fill**

- Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions:
- NONE (default),
- X (fill only horizontally),
- Y (fill only vertically),
- BOTH (fill both horizontally and vertically).

**expand**

- When set to true, widget expands to fill any space not otherwise used in widget's parent.

**ipadx , ipady**

- Internal padding. Default is 0.

**padx, pady**

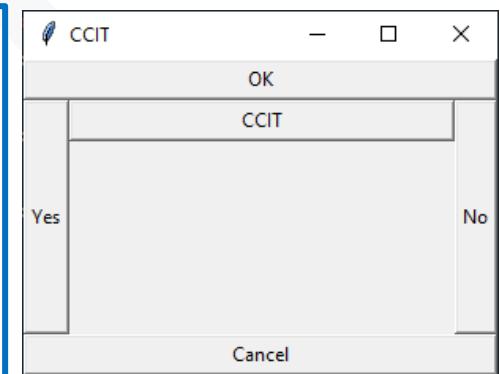
- External padding. Default is 0.

**Example :****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.btn1=Button(self,text="OK")
        self.btn1.pack(fill=X)
        self.btn2=Button(self,text="Cancel")
        self.btn2.pack(side=BOTTOM,fill=X)
        self.btn3=Button(self,text="Yes")
        self.btn3.pack(side=LEFT,fill=Y)
        self.btn4=Button(self,text="No")
        self.btn4.pack(side=RIGHT,fill=Y)
        self.btn5=Button(self,text="CCIT")
        self.btn5.pack(fill=X)

frm=MyFrame()
frm.mainloop()
```

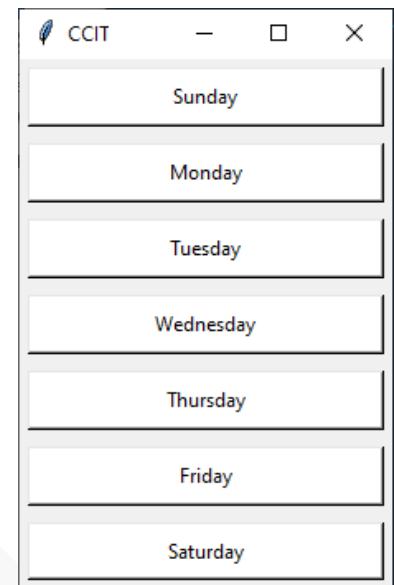
**Output**

**Example:****Program**

```
from tkinter import *

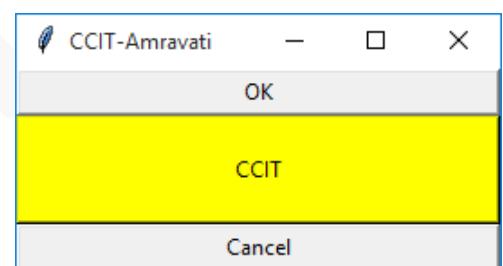
class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        lst=["Sunday", "Monday",
             "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday"]
        for nm in lst:
            self.btn=Button(self, text=nm, bg="white")
            self.btn.pack(fill=X, padx=5, pady=5, ipady=5)

frm=MyFrame()
frm.mainloop()
```

**Output****Example:****Program**

```
from tkinter import *
class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x300")
        self.b1=Button(self, text="OK")
        self.b1.pack(side=TOP, fill=X)
        self.b2=Button(self, text="Cancel")
        self.b2.pack(side=BOTTOM, fill=X)
        self.b3=Button(self, text="CCIT", bg="yellow")
        self.b3.pack(fill=BOTH, expand=True)

frm=MyFrame()
frm.mainloop()
```

**Output**

# Grid managers

Grid is in many cases the best choice for general use. While pack is sometimes not sufficient for changing details in the layout, place gives you complete control of positioning each element, but this makes it a lot more complex than pack and grid.

The Grid geometry manager places the widgets in a 2-dimensional table, which consists of a number of rows and columns. The position of a widget is defined by a row and a column number. Widgets with the same column number and different row numbers will be above or below each other. Correspondingly, widgets with the same row number but different column numbers will be on the same "line" and will be beside of each other, i.e. to the left or the right.

Using the grid manager means that you create a widget, and use the grid method to tell the manager in which row and column to place them. The size of the grid doesn't have to be defined, because the manager automatically determines the best dimensions for the widgets used.

## Syntax:

```
widget.grid( grid_options )
```

**column** - The column to put widget in; default 0 (leftmost column).

**columnspan** - How many columns widget occupies; default 1.

**ipadx, ipady** - How many pixels to pad widget, horizontally and vertically, inside widget's borders.

**padx, pady** - How many pixels to pad widget, horizontally and vertically, outside v's borders.

**row** - The row to put widget in; default the first row that is still empty.

**rowspan** - How many rows widget occupies; default 1.

**sticky** - What to do if the cell is larger than widget. By default, with sticky="", widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

**Example:****Program**

```
import tkinter as gui
win = gui.Tk()
win.title('CCIT')
for r in range(0,4):
    for c in range(0,4):
        b=gui.Button(win,text="CCIT-"+str(r)+","+str(c))
    b.grid(row=r,column=c)

win.mainloop()
```

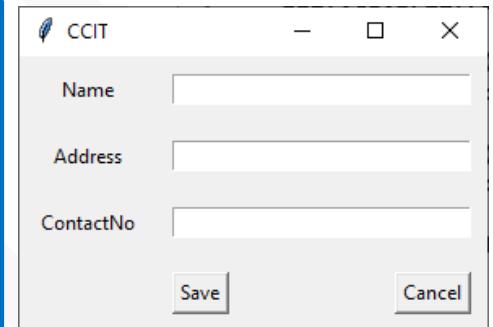
**Output**

CCIT-0,0	CCIT-0,1	CCIT-0,2	CCIT-0,3
CCIT-1,0	CCIT-1,1	CCIT-1,2	CCIT-1,3
CCIT-2,0	CCIT-2,1	CCIT-2,2	CCIT-2,3
CCIT-3,0	CCIT-3,1	CCIT-3,2	CCIT-3,3

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.la=Label(self,text="Name")
        self.la.grid(row=0,column=0,padx=10,pady=10)
        self.ta=Entry(self,width=30)
        self.ta.grid(row=0,column=1,padx=10,pady=10,
        columnspan=2)
        self.lb=Label(self,text="Address")
        self.lb.grid(row=1,column=0,padx=10,pady=10)
        self.tb=Entry(self,width=30)
        self.tb.grid(row=1,column=1,padx=10,pady=10,
        columnspan=2)
        self.lc=Label(self,text="ContactNo")
        self.lc.grid(row=2,column=0,padx=10,pady=10)
        self.tc=Entry(self,width=30)
        self.tc.grid(row=2,column=1,padx=10,pady=10,
        columnspan=2)
        self.b1=Button(self,text="Save")
        self.b1.grid(row=3,column=1,padx=10,pady=10,
        sticky='W')
        self.b2=Button(self,text="Cancel")
        self.b2.grid(row=3,column=2,padx=10,pady=10,
        sticky='E')
        self.title("CCIT")
frm=MyFrame()
frm.mainloop()
```

**Output**

# Place managers

The Place geometry manager allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window. The place manager can be accessed through the place method. It can be applied to all standard widgets.

We use the place geometry manager in the following example. We are playing around with colours in this example, i.e. we assign to every label a different colour, which we randomly create using the randrange method of the random module. We calculate the brightness (grey value) of each colour. If the brightness is less than 120, we set the foreground colour (fg) of the label to White otherwise to black, so that the text can be easier read.

## Syntax:

```
widget.place( place_options )
```

**anchor** – The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW (the upper left corner of widget)

**bordermode** – INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.

**height, width** – Height and width in pixels.

**relheight, relwidth** – Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.

**relx, rely** – Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.

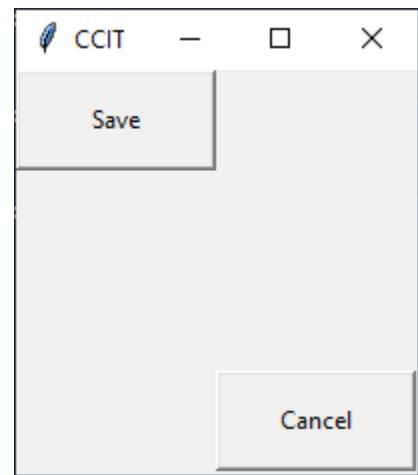
**x, y** – Horizontal and vertical offset in pixels.

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("200x200")
        self.b1=Button(self,text="Save")
        self.b1.place(x=0,y=0,width=100,height=50)
        self.b2=Button(self,text="Cancel")
        self.b2.place(x=100,y=150,width=100,height=50)
        self.title('CCIT')

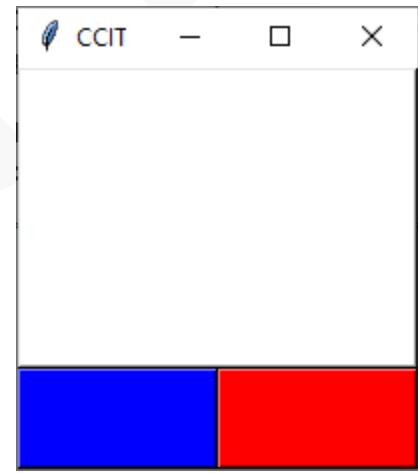
frm=MyFrame()
frm.mainloop()
```

**Output****Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("200x200")
        self.title('CCIT')
        self.b1=Button(self,bg="white")
        self.b1.place(relx=0.0,rely=0.0,relwidth=1.0,
        relheight=0.75)
        self.b2=Button(self,bg="blue")
        self.b2.place(relx=0.0,rely=0.75,relwidth=0.5,
        relheight=0.25)
        self.b3=Button(self,bg="red")
        self.b3.place(relx=0.5,rely=0.75,relwidth=0.5,
        relheight=0.25)

frm=MyFrame()
frm.mainloop()
```

**Output**

# Spinbox

The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values. The Spinbox widget is an alternative to the Entry widget. It provides the range of values to the user, out of which, the user can select the one. It is used in the case where a user is given some fixed number of values to choose from.

## Syntax:

```
Spinbox(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>delete(startindex, endindex)</b>	This method is used to delete the characters present at the specified range.
<b>get(startindex, endindex)</b>	Deselects the checkbox; that is, sets the value to offvalue.
<b>insert(index, string)</b>	This method inserts strings at the specified index location.
<b>index(index)</b>	Returns the absolute value of an index based on the given index.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure methods

Attribute	Description
<b>from_</b>	The minimum value. Used together with to to limit the spinbox range.
<b>to</b>	See from.
<b>increment</b>	It sets increment by value.
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.

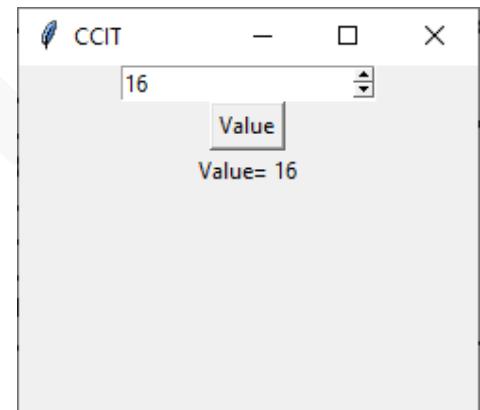
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>command</b>	A procedure to be called every time the user changes the state of this checkbutton.
<b>font</b>	Text font to be used for the button's label.
<b>textvariable</b>	It is like a control variable which is used to control the behaviour of the widget text.
<b>state</b>	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.title("CCIT")
        self.spin=Spinbox(self,from_=0,to=20,increment=
2)
        self.spin.pack()
        self.btn=Button(self,text="Value",command=
self.val)
        self.btn.pack()
        self.lbl=Label(self,)
        self.lbl.pack()
    def val(self):
        self.lbl.configure(text="Value="+self.spin.get())

frm=MyFrame()
frm.mainloop()
```

**Output**

# Scale

The Scale widget is used to implement the graphical slider to the python application so that the user can slide through the range of values shown on the slider and select the one among them.

We can control the minimum and maximum values along with the resolution of the scale. It provides an alternative to the Entry widget when the user is forced to select only one value from the given range of values.

## Syntax:

```
Scale(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>get()</b>	This method returns the current value of the scale.
<b>set ( value )</b>	Sets the scale's value.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure methods

Attribute	Description
<b>from_</b>	The minimum value. Used together with to to limit the spinbox range.
<b>to</b>	It represents a float or integer value that specifies the other end of the range represented by the scale.
<b>resolution</b>	It is set to the smallest change which is to be made to the scale value.
<b>orient</b>	It can be set to horizontal or vertical depending upon the type of the scale.
<b>sliderlength</b>	It represents the length of the slider window along the length of the scale. The default is 30 pixels. However, we can change it to the appropriate value.
<b>background ,bg</b>	background color

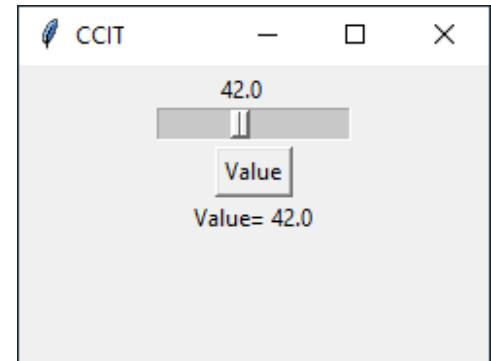
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>command</b>	A procedure to be called every time the user changes the state of this checkbutton.
<b>font</b>	Text font to be used for the button's label.
<b>label</b>	This can be set to some text which can be shown as a label with the scale. It is shown in the top left corner if the scale is horizontal or the top right corner if the scale is vertical.
<b>variable</b>	It represents the control variable for the scale.
<b>state</b>	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.title("CCIT")
        self.spin=Scale(self,from_=0,to=100,
resolution=0.1,sliderlength=10,orient='horizontal')
        self.spin.pack()
        self.btn=Button(self,text="Value",command=
self.val)
        self.btn.pack()
        self.lbl=Label(self,)
        self.lbl.pack()
    def val(self):
        self.lbl.configure(text="Value= "+str(
self.spin.get()))

frm=MyFrame()
frm.mainloop()
```

**Output**

# Checkboxes

Checkboxes, also known as tickboxes or tick boxes or check boxes, are widgets that permit the user to make multiple selections from a number of different options. Usually, checkboxes are shown on the screen as square boxes that can contain white spaces (for false, i.e not checked) or a tick mark or X (for true, i.e. checked).

A caption describing the meaning of the checkbox is usually shown adjacent to the checkbox. The state of a checkbox is changed by clicking the mouse on the box. Alternatively it can be done by clicking on the caption, or by using a keyboard shortcut, for example, the space bar.

A Checkbox has two states: on or off.

The Tkinter Checkbutton widget can contain text, but only in a single font, or images, and a button can be associated with a Python function or method. When a button is pressed, Tkinter calls the associated function or method. The text of a button can span more than one line.

## Syntax:

```
Checkbutton(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>select()</b>	Selects the button; that is, sets the value to onvalue.
<b>deselect()</b>	Deselects the checkbox; that is, sets the value to offvalue.
<b>toggle()</b>	Clears the checkbutton if set, sets it if cleared.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure methods

Attribute	Description
<b>text</b>	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>image</b>	To display a static image in the label widget, set this option to an image object.
<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>underline</b>	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
<b>command</b>	A procedure to be called every time the user changes the state of this checkbutton.
<b>font</b>	Text font to be used for the button's label.
<b>padx</b>	Sets padding x-axis
<b>pady</b>	Sets padding y-axis
<b>variable</b>	The control variable that tracks the current state of the checkbutton. Normally this variable is an IntVar, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
<b>state</b>	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
<b>offvalue</b>	Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting offvalue to that value.
<b>onvalue</b>	Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting onvalue to that value.

**Example:****Program**

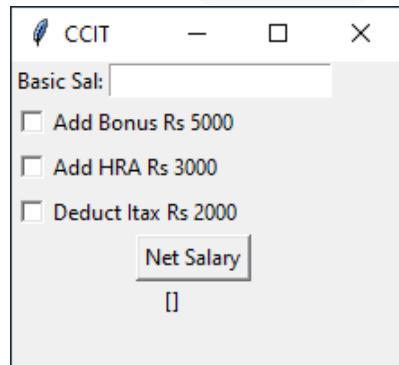
```

from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.geometry("200x200")
        self.la=Label(self,text="Basic Sal:")
        self.la.grid()
        self.ta=Entry(self)
        self.ta.grid(row=0,column=1,columnspan=2)
        self.bonus=IntVar()
        self.c1=Checkbutton(self,text="Add Bonus Rs 5000",onvalue=5000,offvalue=0,
variable=self.bonus)
        self.c1.grid(row=1,column=0,columnspan=3,sticky="W")
        self.hra=IntVar()
        self.c2=Checkbutton(self,text="Add HRA Rs 3000",onvalue=3000,offvalue=0,
variable=self.hra)
        self.c2.grid(row=2,column=0,columnspan=3,sticky="W")
        self.itax=IntVar()
        self.c3=Checkbutton(self,text="Deduct Itax Rs 2000",onvalue=2000,offvalue=
0,variable=self.itax)
        self.c3.grid(row=3,column=0,columnspan=3,sticky="W")
        self.b1=Button(self,text="Net Salary",command=self.cal)
        self.b1.grid(row=4,column=1)
        self.lmsg=Label(self,text="[]")
        self.lmsg.grid(row=5,column=0,columnspan=3)
    def cal(self):
        bs=int(self.ta.get())
        ns=bs+self.bonus.get()+self.hra.get()-self.itax.get()
        self.lmsg.config(text="Net Salary is "+str(ns))

frm=MyFrame()
frm.mainloop()

```

**Output**

# Radio Buttons

A radio button, sometimes called option button, is a graphical user interface element of Tkinter, which allows the user to choose (exactly) one of a predefined set of options. Radio buttons can contain text or images. The button can only display text in a single font. A Python function or method can be associated with a radio button. This function or method will be called, if you press this radio button.

Radio buttons are named after the physical buttons used on old radios to select wave bands or preset radio stations. If such a button was pressed, other buttons would pop out, leaving the pressed button the only pushed in button.

Each group of Radio button widgets has to be associated with the same variable. Pushing a button changes the value of this variable to a predefined certain value.

## Syntax:

```
Radiobutton(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>select()</b>	Selects the button; that is, sets the value to onvalue.
<b>deselect()</b>	Deselects the checkbox; that is, sets the value to offvalue.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

Attribute	Description
<b>text</b>	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.

<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>image</b>	To display a static image in the label widget, set this option to an image object.
<b>justify</b>	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
<b>underline</b>	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>font</b>	Text font to be used for the button's label.
<b>padx</b>	Sets padding x-axis
<b>pady</b>	Sets padding y-axis
<b>variable</b>	The control variable that tracks the current state of the checkbutton. Normally this variable is an IntVar, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
<b>state</b>	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
<b>relief</b>	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, flat ,and RIDGE.
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>value</b>	When a radiobutton is turned on by the user, its control variable is set to its current value option. If the control variable is an IntVar, give each radiobutton in the group a different integer value option. If the control variable is a StringVar, give each radiobutton a different string value option.
<b>selectcolor</b>	The color of the radiobutton when it is set. Default is red.
<b>command</b>	A procedure to be called every time the user changes the state of this radiobutton.

## Example:

### Program

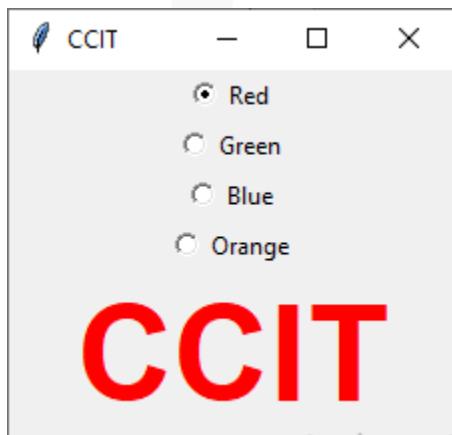
```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        lst=["Red","Green","Blue","Orange"]
        self.color=StringVar()
        self.color.set("Red")
        for clr in lst:
            self.rbtn=Radiobutton(self,text=clr,value=clr,variable=self.color,
command=self.show)
            self.rbtn.pack()
        self.lbl=Label(self,text="CCIT",fg="Red", font="Helvetica 50 bold")
        self.lbl.pack()

    def show(self):
        clr=self.color.get()
        self.lbl.config(fg=clr)

frm=MyFrame()
frm.mainloop()
```

### Output



# Listbox

The Listbox widget is a standard Tkinter widget used to display a list of alternatives. The listbox can only contain text items, and all items must have the same font and color. Depending on the widget configuration, the user can choose one or more alternatives from the list.

Listboxes are used to select from a group of textual items. Depending on how the listbox is configured, the user can select one or many items from that list.

## Syntax:

```
Listbox(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>activate(index)</b>	Selects the line specifies by the given index.
<b>curselection()</b>	Returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
<b>delete(first, last=None)</b>	Deletes the lines whose indices are in the range [first, last]. If the second argument is omitted, the single line with index first is deleted.
<b>get(first, last=None)</b>	Returns a tuple containing the text of the lines with indices from first to last, inclusive. If the second argument is omitted, returns the text of the line closest to first.
<b>index(i)</b>	If possible, positions the visible part of the listbox so that the line containing index i is at the top of the widget.
<b>insert(index, *elements )</b>	Insert one or more new lines into the listbox before the line specified by index. Use END as the first argument if you want to add new lines to the end of the listbox.
<b>size()</b>	Returns the number of lines in the listbox.
<b>see ( index )</b>	Adjust the position of the listbox so that the line referred to by index is visible.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).

<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

<b>Attribute</b>	<b>Description</b>
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>font</b>	Text font to be used for the button's label.
<b>selectbackground</b>	The background color to use displaying selected text.
<b>selectmode</b>	Determines how many items can be selected, and how mouse drags affect the selection – <ul style="list-style-type: none"> <li>• BROWSE – Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default.</li> <li>• SINGLE – You can only select one line, and you can't drag the mouse wherever you click button 1, that line is selected.</li> <li>• MULTIPLE – You can select any number of lines at once. Clicking on any line toggles whether or not it is selected.</li> <li>• EXTENDED – You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.</li> </ul>
<b>xscrollcommand</b>	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
<b>yscrollcommand</b>	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.
<b>xview</b>	Query and change the horizontal position of the view.
<b>yview</b>	Query and change the vertical position of the view.

## Example:

### Program

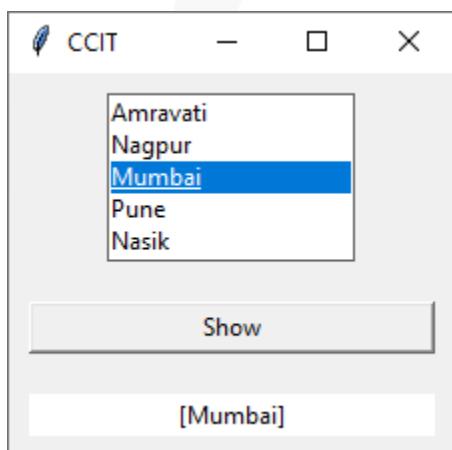
```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.lst=Listbox(self,height=5)
        self.lst.pack(padx=10,pady=10)
        self.lst.configure(relief='flat')
        self.lst.insert(0,"Amravati","Nagpur","Mumbai","Pune","Nasik","Akola")
        self.btn=Button(self,text="Show",command=self.show)
        self.btn.pack(fill=X,padx=10,pady=10)
        self.lbl=Label(self,text="[ ]",bg="white")
        self.lbl.pack(fill=X,padx=10,pady=10)

    def show(self):
        tp=self.lst.curselection()
        item=self.lst.get(tp[0])
        self.lbl.config(text="[ "+item+" ]")

frm=MyFrame()
frm.mainloop()
```

### Output



## Example:

### Program

```
from tkinter import *

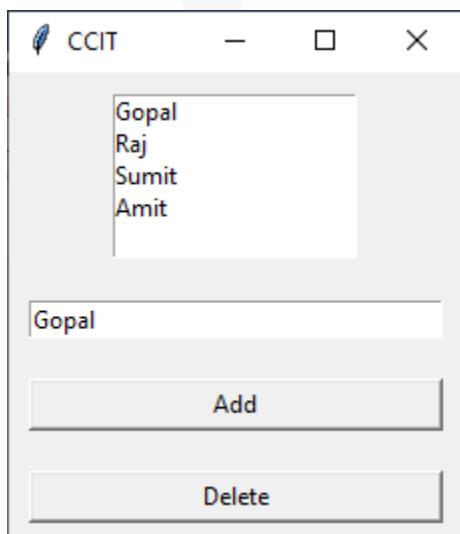
class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.data=StringVar()
        self.lst=Listbox(self,height=5)
        self.lst.pack(padx=10,pady=10)
        self.txt=Entry(self,textvariable=self.data)
        self.txt.pack(fill=X,padx=10,pady=10)
        self.btn=Button(self,text="Add",command=self.add)
        self.btn.pack(fill=X,padx=10,pady=10)
        self.btnx=Button(self,text="Delete",command=self.delete)
        self.btnx.pack(fill=X,padx=10,pady=10)

    def add(self):
        self.lst.insert(0,self.data.get())
        self.data.set("")

    def delete(self):
        tp=self.lst.curselection()
        self.lst.delete(tp[0])

frm=MyFrame()
frm.mainloop()
```

### Output



# Canvas

The Canvas widget is Tkinter's primary widget for displaying graphics. With a vast range of built-in functions for creating graphics manually, it is the perfect choice for the display piece of a computer game.

The Canvas widget handles coordinates with a Cartesian system, with the origin in the top-left of the window. The Y coordinate will go down the window as its value increases, which may take some getting used to if you are familiar with other software in which a positive Y value instead goes upwards. The X coordinate goes further right as it increases, as with most other systems.

## Syntax:

```
Canvas(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<code>create_rectangle(*args[x1,y1,x2,y2], **kw[width,fill,outline])</code>	Create rectangle with coordinates x1,y1,x2,y2.
<code>create_oval(*args[x1,y1,x2,y2], **kw[width,fill,outline])</code>	Create oval with coordinates x1,y1,x2,y2.
<code>create_polygon(*args[x1,y1,x2,y2,...xn,yn], **kw[width,fill,outline])</code>	Create polygon with coordinates x1,y1,...,xn,yn.
<code>create_arc(*args[x1,y1,x2,y2], **kw[start,extent,width,fill,outline])</code>	Create arc shaped region with coordinates x1,y1,x2,y2.
<code>create_line( *args[x1,y1,x2,y2,...xn,yn], **kw[fill,width])</code>	Create line with coordinates x1,y1,...,xn,yn.
<code>create_text(*args[x,y], **kw[text,fill,font])</code>	Create text with coordinates x1,y1.
<code>create_image(x,y, anchor=pos, image=imgObj)</code>	To display image
<code>pack( )</code>	To add widget into container(window).
<code>place( )</code>	To add widget into container(window).

<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

Where

- width : indicates width of outline in pixels
- fill : indicates fill color
- outline : indicates color of outline

## configure method attributes

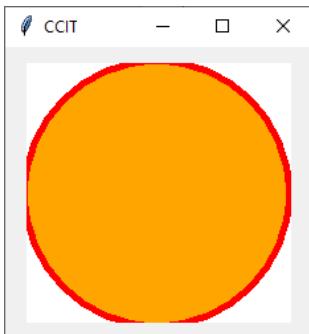
Attribute	Description
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>fg</b>	The color used to render the text.
<b>width</b>	Sets width
<b>height</b>	Sets height
<b>cursor</b>	Curosr for window Values: "arrow" , "circle" , "clock" , "cross" , "plus" , "watch" etc
<b>relief</b>	It represents the type of the border. The possible values are SUNKEN, RAISED, GROOVE, and RIDGE.
<b>font</b>	Text font to be used for the button's label.
<b>scrollregion</b>	A tuple (w, n, e, s) that defines over how large an area the canvas can be scrolled, where w is the left side, n the top, e the right side, and s the bottom.
<b>xscrollcommand</b>	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
<b>xview</b>	Query and change the horizontal position of the view.
<b>yscrollcommand</b>	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.
<b>yview</b>	Query and change the vertical position of the view.

**Example:****Program**

```
from tkinter import *

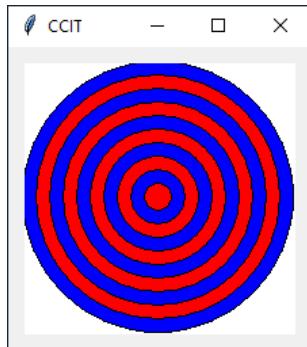
class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.can=Canvas(self,width=200,height=200,bg="white")
        self.can.pack(padx=10,pady=10)
        self.can.create_oval([0,0,200,200],fill="orange",outline="red",width=5)

frm=MyFrame()
frm.mainloop()
```

**Output****Example:****Program**

```
from tkinter import *
class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.can=Canvas(self,width=200,height=200,bg="white")
        self.can.pack(padx=10,pady=10)
        j=1
        for i in range(0,100,10):
            x1=i
            y1=i
            x2=200-i
            y2=200-i
            if j%2==0:
                self.can.create_oval([x1,y1,x2,y2],fill="red")
            else:
                self.can.create_oval([x1,y1,x2,y2],fill="blue")
            j+=1

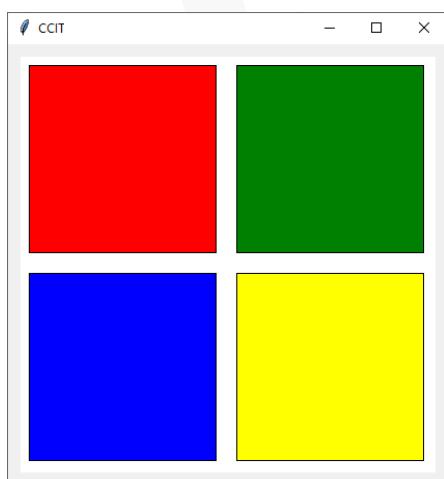
frm=MyFrame()
frm.mainloop()
```

**Output****Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.title('CCIT')
        self.can=Canvas(self,width=400,height=400,bg="white")
        self.can.pack(padx=10,pady=10)
        self.can.create_rectangle([10,10,190,190],fill="red")
        self.can.create_rectangle([210,10,390,190],fill="green")
        self.can.create_rectangle([10,210,190,390],fill="blue")
        self.can.create_rectangle([210,210,390,390],fill="yellow")

frm=MyFrame()
frm.mainloop()
```

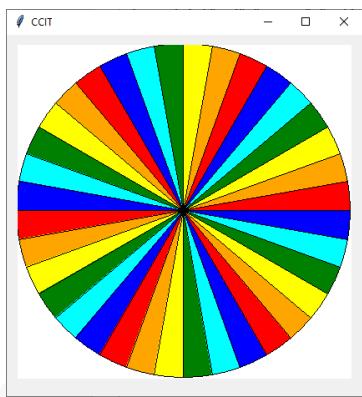
**Output**

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.can=Canvas(self,width=400,height=400,bg="white")
        self.can.pack(padx=10,pady=10)
        lst=["red","orange","yellow","green","cyan","blue"]
        for i in range(0,36):
            self.can.create_arc([0,0,400,400],start=i*10,extent=10,fill=lst[i%6])

frm=MyFrame()
frm.mainloop()
```

**Output**

# Dialogues

Tkinter provides a set of dialogues, which can be used to display message boxes, showing warning or errors, or widgets to select files and colours. There are also simple dialogues, asking the user to enter string, integers or float numbers.

## Message boxes

The `tkinter.messagebox` module provides a template base class as well as a variety of convenience methods for commonly used configurations. The message boxes are modal and will return a subset of (True, False, OK, None, Yes, No) based on the users selection. A messagebox can display information to a user. There are three variations on these dialog boxes based on the type of message you want to display.

### Information message box

- `tkinter.messagebox.showinfo(title=None, message=None, **options)`

### Warning message boxes

- `tkinter.messagebox.showwarning(title=None, message=None, **options)`
- `tkinter.messagebox.showerror(title=None, message=None, **options)`

### Question message boxes

- `tkinter.messagebox.askquestion(title=None, message=None, **options)`
- `tkinter.messagebox.askokcancel(title=None, message=None, **options)`
- `tkinter.messagebox.askretrycancel(title=None, message=None, **options)`
- `tkinter.messagebox.askyesno(title=None, message=None, **options)`
- `tkinter.messagebox.askyesnocancel(title=None, message=None, **options)`

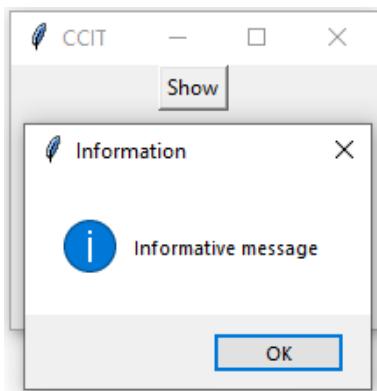
### Example:

#### Program

```
import tkinter as tk
from tkinter import *
from tkinter import messagebox

def show():
    messagebox.showinfo("Information", "Informative message")
    messagebox.showerror("Error", "Error message")
    messagebox.showwarning("Warning", "Warning message")

win=tk.Tk()
win.title('CCIT')
win.geometry("150x150")
btn=Button(win,text="Show",command=show)
btn.pack()
win.mainloop()
```

**Output****Example:****Program**

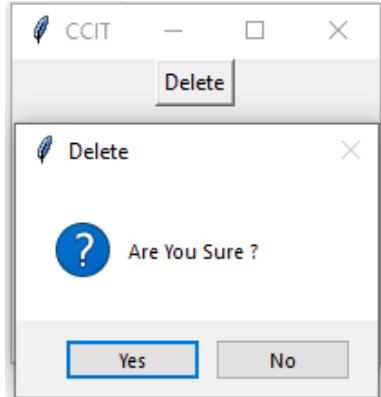
```
import tkinter as tk
from tkinter import *
from tkinter import messagebox

def show():
    res=messagebox.askyesno("Delete","Are You Sure ?")
    if res==True:
        messagebox.showinfo("Delete","Kar Deya")

win=tk.Tk()
win.title('CCIT')
win.geometry("150x250")

btn=Button(win,text="Delete",command=show)
btn.pack()

win.mainloop()
```

**Output**

# simpledialog

The `tkinter.simpaledialog` module contains convenience classes and functions for creating simple modal dialogs to get a value from the user. If you want to ask the user for a single data value, either a string, integer, or floating point value, you can use a `simpaledialog` object. A user can enter the requested value and hit "OK", which will return the entered value. If the user hits "Cancel," then `None` is returned.

- `tkinter.simpaledialog.askfloat(title, prompt, **kw)`
- `tkinter.simpaledialog.askinteger(title, prompt, **kw)`
- `tkinter.simpaledialog.askstring(title, prompt, **kw)`

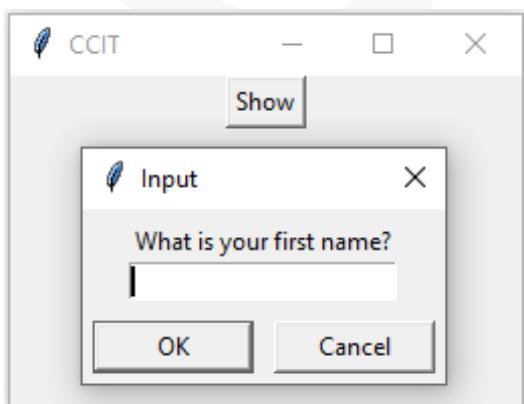
## Example:

### Program

```
from tkinter import *
from tkinter import simpaledialog

class Win(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("150x250")
        self.title("CCIT")
        self.btn=Button(self,text="Show",command=self.show)
        self.btn.pack()
    def show(self):
        answer = simpaledialog.askstring("Input", "What is your first name?")
        if answer is not None:
            print("Your first name is ", answer)
        else:
            print("You don't have a first name?")
win=Win()
win.mainloop()
```

### Output



# File Choose

A common task is to select the names of folders and files on a storage device. This can be accomplished using a filedialog object. Note that these commands do not save or load a file. They simply allow a user to select a file. Once you have the file name, you can open, process, and close the file using appropriate Python code. These dialog boxes always return you a “fully qualified file name” that includes a full path to the file. Also note that if a user is allowed to select multiple files, the return value is a tuple that contains all of the selected files. If a user cancels the dialog box, the returned value is an empty string.

## Open Files

- **`tkinter.filedialog.askopenfile(mode="r", **options)`**
- **`tkinter.filedialog.askopenfiles(mode="r", **options)`**

The above two functions create an Open dialog and return the opened file object(s) in read-only mode.

- **`tkinter.filedialog.askopenfilename(**options)`**
- **`tkinter.filedialog.askopenfilenames(**options)`**

The above two functions create an Open dialog and return the selected filename(s) that correspond to existing file(s).

## Save Files

- **`tkinter.filedialog.asksaveasfile(mode="w", **options)`**

Create a SaveAs dialog and return a file object opened in write-only mode.

- **`tkinter.filedialog.asksaveasfilename(**options)`**

Create a SaveAs dialog and return the selected filename.

## Open Folder

- **`tkinter.filedialog.askdirectory(**options)`**

Prompt user to select a directory.

Options can be

- title - a title for dialogbox
- initialdir – a initial dir to be shown in dialog box
- filetypes – file filter for dialogbox

For ex [('all files', '\*'), ('text files', '.txt'), ('Images', '.gif; .png')]

## Example:

### Program

```
import tkinter as tk
from tkinter import *
from tkinter import filedialog
import os

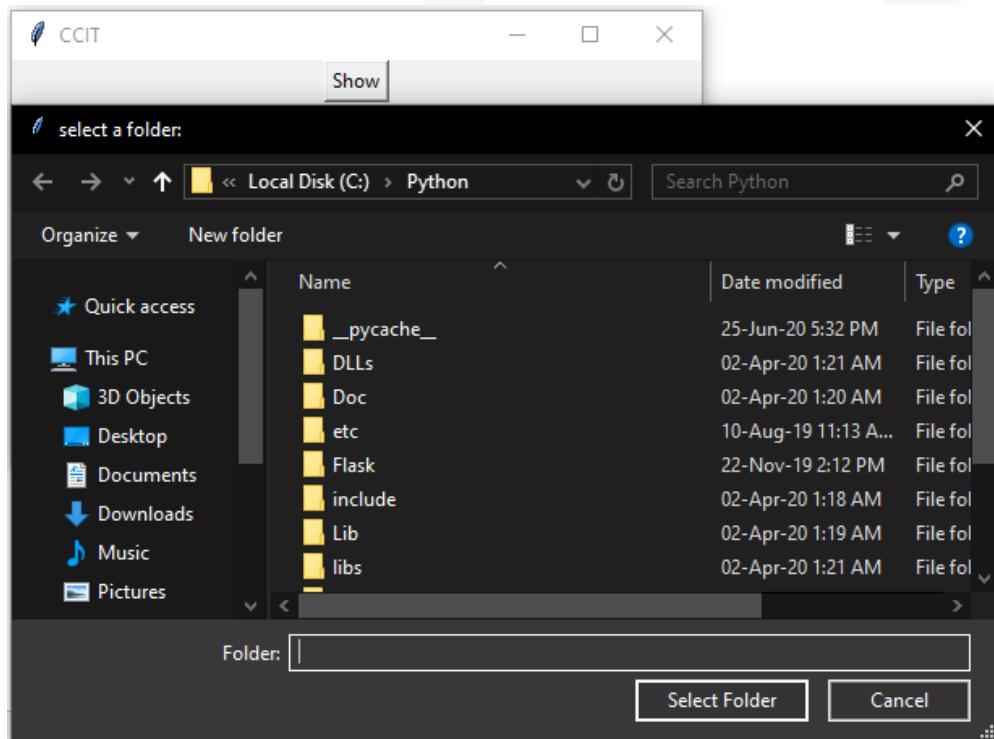
def show():
    ans = filedialog.askdirectory(initialdir=os.getcwd(), title="select folder:")
    print(ans)

win=tk.Tk()
win.title("CCIT")
win.geometry("150x250")

btn=Button(win, text="Show", command=show)
btn.pack()

win.mainloop()
```

### Output



## Example:

### Program

```
import tkinter as tk
from tkinter import *
from tkinter import filedialog
import os

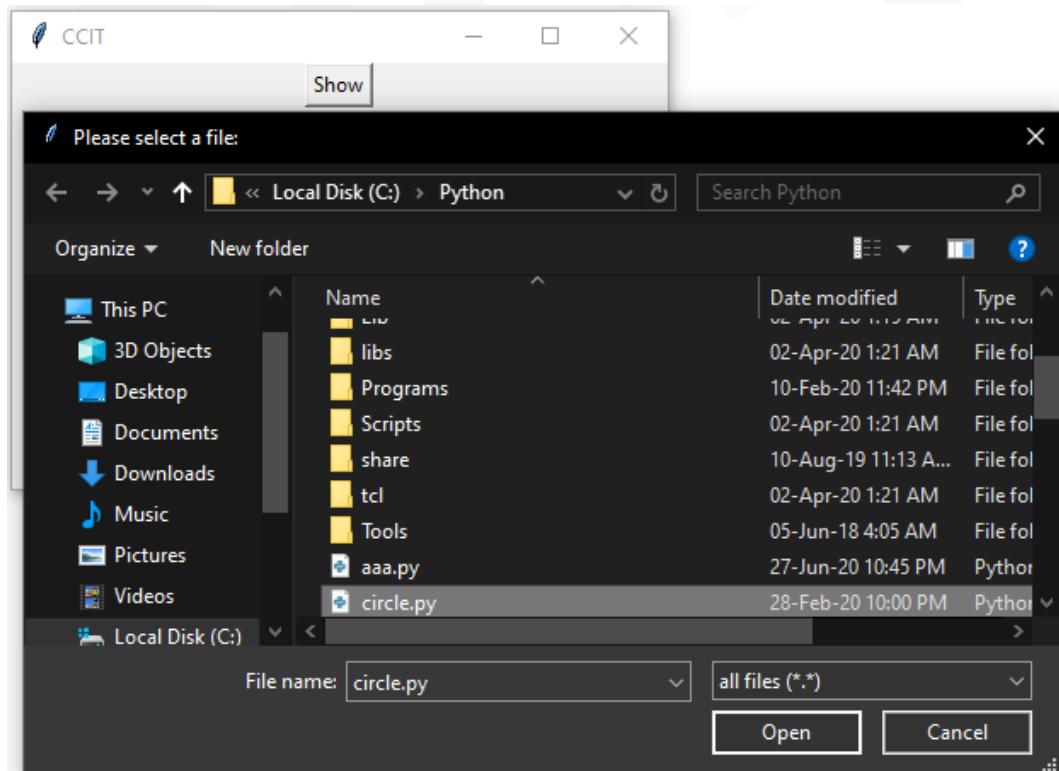
def show():
    my_filetypes = [ ('all files', '*'), ('text files', '.txt')]
    ans = filedialog.askopenfilename(initialdir=os.getcwd(),title="Please
select a file:",filetypes=my_filetypes)
    print(ans)

win=tk.Tk()
win.title("CCIT")
win.geometry("150x250")

btn=Button(win,text="Show",command=show)
btn.pack()

win.mainloop()
```

### Output



# Color Chooser

Tkinter includes a nice dialog box for choosing colors. You provide it with a parent window and an initial color, and it returns a color in two different specifications: 1) a RGB value as a tuple, such as (255, 0, 0) which represents red, and 2) a hexadecimal string used in web pages, such as "#FF0000" which also represents red. If the user cancels the operation, the return values are None and None.

## Syntax:

```
tkinter.colorchooser.askcolor(color=None, **options)
```

## Example:

### Program

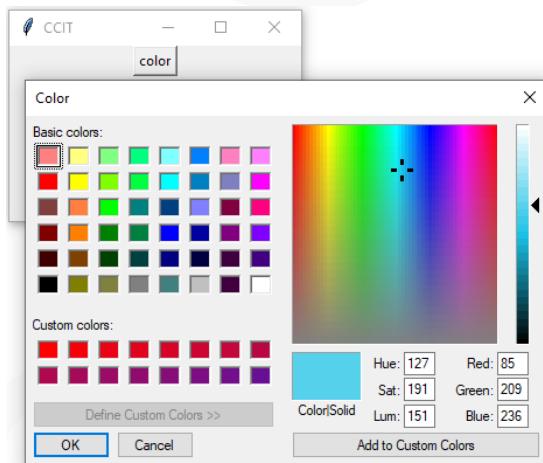
```
import tkinter as tk
from tkinter import *
from tkinter import colorchooser

class Win(Tk):
    def __init__(self):
        super().__init__()
        self.title("CCIT")
        self.geometry("250x150")
        self.btn=Button(self,text="color",command=self.show)
        self.btn.pack()

    def show(self):
        color = colorchooser.askcolor()
        self.configure(background=color[1])

win=Win()
win.mainloop()
```

### Output



# Class PhotoImage

It is used to create an image object. The PhotoImage class can read GIF and PNG images from files. The PhotoImage class is used to display images (either grayscale or true color images) in labels, buttons, canvases, and text widgets. You can use the PhotoImage class whenever you need to display an icon or an image in a Tkinter application. PhotoImage for images in PGM, PPM, GIF and PNG formats.

## Syntax:

```
PhotoImage(name=None, cnf={}, master=None, **kw)
```

## object methods

An instance of the class supplies the following methods

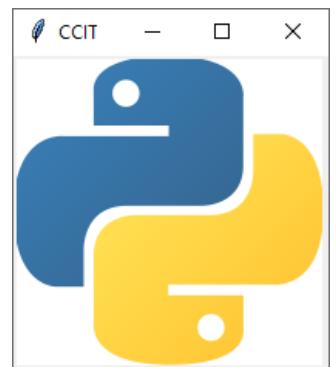
Method	Description
<b>zoom( x, y= '' )</b>	Return a new PhotoImage with the same image as this widget but zoom it with a factor of x in the X direction and y in the Y direction. If y is not given, the default value is the same as x.
<b>get( x, y )</b>	Return the color (red, green, blue) of the pixel at X,Y.
<b>put(self, data, to=None)</b>	Put row formatted colors to image starting from position TO, e.g. image.put("{red green} {blue yellow}", to=(4,6))
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attribute

Attribute	Description
<b>file</b>	Image file name
<b>height</b>	Sets height
<b>width</b>	Sets width

**Example:****Program**

```
from tkinter import *
root = Tk()
root.title("CCIT")
canvas = Canvas(root, width = 200, height = 200)
canvas.pack()
img = PhotoImage(file="python2.png",height=500,width=500)
canvas.create_image(00,00, anchor=NW, image=img)
root.mainloop()
```

**Output****Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        Tk.__init__(self)
        self.title("CCIT")
        self.iname=StringVar()
        self.iname.set("r")
        rb1=Radiobutton(self,text="python",variable=
        self.iname, value="python2", command=self.show)
        rb1.grid(row=0,column=0)
        rb2=Radiobutton(self,text="java",variable= self.iname,
        value="java", command=self.show)
        rb2.grid(row=0,column=1)
        rb3=Radiobutton(self,text="c++",variable= self.iname,
        value="cpp", command=self.show)
        rb3.grid(row=0,column=2)
        self.can=Canvas(self,width=250,height=500)
        self.can.grid(row=1,column=0,columnspan=3)
        self.img=PhotoImage(file="python2.png")
        self.can.create_image(20,20,anchor=NW,image=
        self.img)
    def show(self):
        fname=self.iname.get()
        self.img=PhotoImage(file=fname+".png")
        self.can.create_image(20,20,anchor=NW,
        image=self.img)

win=MyFrame()
win.mainloop()
```

**Output**

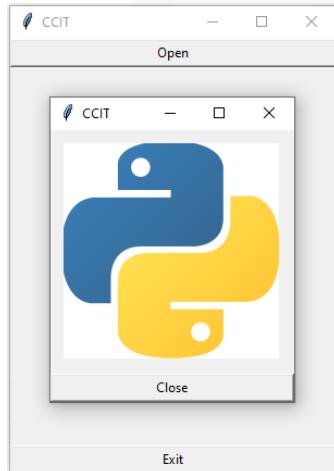
**Example:****Program**

```

from tkinter import *
from tkinter import filedialog
class ImgDialog(Toplevel):
    def __init__(self,fnm):
        super().__init__()
        self.can=Canvas(self,width=200,height=200)
        self.can.pack(padx=10,pady=10)
        self.img=PhotoImage(file=fnm)
        self.can.create_image(0,0,anchor=NW,image=self.img)
        self.btn=Button(self,text="Close",command=self.close)
        self.btn.pack(fill=X)
    def close(self):
        self.destroy()
class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x400")
        self.b1=Button(self,text="Open",command=self.open)
        self.b1.pack(fill=X)
        self.b2=Button(self,text="Exit",command=self.exit)
        self.b2.pack(side=BOTTOM,fill=X)
    def open(self):
        ftypes=[("Images",".gif; .png")]
        fnm=filedialog.askopenfilename(filetypes=filetypes,initialdir="c:/temp")
        if fnm!="":
            dlg=ImgDialog(fnm)
            dlg.grab_set()
            self.wait_window(dlg)
    def exit(self):
        self.destroy()

frm=MyFrame()
frm.mainloop()

```

**Output**

# Menus

Menus in GUIs are presented with a combination of text and symbols to represent the choices. Selecting with the mouse (or finger on touch screens) on one of the symbols or text, an action will be started. Such an action or operation can, for example, be the opening or saving of a file, or the quitting or exiting of an application.

A context menu is a menu in which the choices presented to the user are modified according to the current context in which the user is located.

The Menu widget is used to create various types of menus (top level, pull down, and pop up) in the python application.

The top-level menus are the one which is displayed just under the title bar of the parent window. We need to create a new instance of the Menu widget and add various commands to it by using the add() method.

## Syntax:

```
Menu(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>add_command(options)</b>	Adds a menu item to the menu.
<b>add_radiobutton(options)</b>	Creates a radio button menu item.
<b>add_checkbutton(options)</b>	Creates a check button menu item.
<b>add_cascade(options)</b>	Adds a submenu.
<b>add_separator()</b>	Adds a separator line.
<b>delete(startindex, endindex)</b>	Deletes the menu items ranging from startindex to endindex.
<b>type(index)</b>	Returns the type of the choice specified by index: either "cascade", "checkbutton", "command", "radiobutton", "separator", or "tearoff".
<b>add( type, options )</b>	Adds a specific type of menu item to the menu.
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

Attribute	Description
<b>activebackground</b>	The background color that will appear on a choice when it is under the mouse.
<b>activeborderwidth</b>	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel.
<b>activeforeground</b>	The foreground color that will appear on a choice when it is under the mouse.
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>height</b>	Sets height
<b>width</b>	Sets width
<b>font</b>	The default font for textual choices.
<b>image</b>	To display an image on this menubutton.
<b>selectcolor</b>	Specifies the color displayed in checkboxes and radiobuttons when they are selected.
<b>cursor</b>	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off.

### Example:

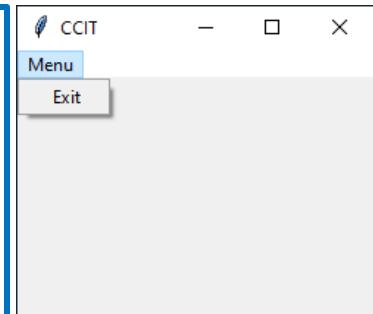
#### Program

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.mbr=Menu(self)
        self.config(menu=self.mbr)
        self.fmenu=Menu(self.mbr,tearoff=0)
        self.fmenu.add_command(label="Exit",command=self.exit)
        self.mbr.add_cascade(label="Menu",menu=self.fmenu)
    def exit(self):
        self.destroy()

win=MyFrame()
win.mainloop()
```

#### Output



**Example:****Program**

```

from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")

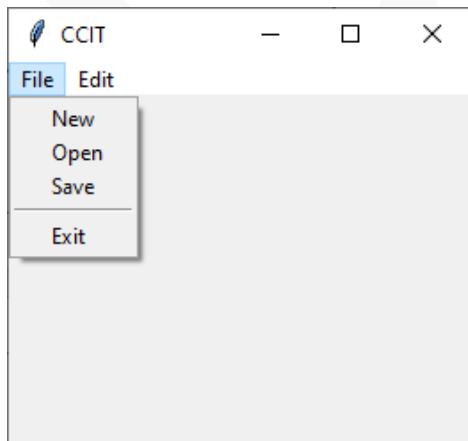
        self.mbar=Menu(self)
        self.config(menu=self.mbar)

        self.fmenu=Menu(self.mbar,tearoff=0)
        self.fmenu.add_command(label="New")
        self.fmenu.add_command(label="Open")
        self.fmenu.add_command(label="Save")
        self.fmenu.add_separator()
        self.fmenu.add_command(label="Exit",command=self.exit)
        self.mbar.add_cascade(label="File",menu=self.fmenu)

        self.emenu=Menu(self.mbar,tearoff=0)
        self.emenu.add_command(label="Undo")
        self.emenu.add_separator()
        self.emenu.add_command(label="Cut")
        self.emenu.add_command(label="Copy")
        self.emenu.add_command(label="Paste")
        self.emenu.add_command(label="Select All")
        self.mbar.add_cascade(label="Edit",menu=self.emenu)
    def exit(self):
        self.destroy()

frm=MyFrame()
frm.mainloop()

```

**Output**

# Text

A text widget is used for multi-line text area. The tkinter text widget is very powerful and flexible and can be used for a wide range of tasks. Though one of the main purposes is to provide simple multi-line areas, as they are often used in forms, text widgets can also be used as simple text editors or even web browsers.

The Text widget is used to show the text data on the Python application. However, Tkinter provides us the Entry widget which is used to implement the single line text box.

The Text widget is used to display the multi-line formatted text with various styles and attributes. The Text widget is mostly used to provide the text editor to the user.

The Text widget also facilitates us to use the marks and tabs to locate the specific sections of the Text. We can also use the windows and images with the Text as it can also be used to display the formatted text.

## Syntax:

```
Text(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>delete(startindex ,endindex)</b>	This method deletes a specific character or a range of text.
<b>get(startindex ,endindex)</b>	This method returns a specific character or a range of text.
<b>insert(index ,string)</b>	This method inserts strings at the specified index location.
<b>tag_add(tagname, startindex,endindex ...)</b>	This method tags either the position defined by startindex, or a range delimited by the positions startindex and endindex.
<b>tag_configure(tagName, options...)</b>	You can use this method to configure the tag properties, which include, justify(center, left, or right), tabs, and underline
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

Attribute	Description
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>height, width</b>	Sets height , Sets width
<b>font</b>	The default font for textual choices.
<b>padx</b>	The size of the internal padding added to the left and right of the text area. Default is one pixel.
<b>pady</b>	The size of the internal padding added above and below the text area. Default is one pixel.
<b>selectbackground</b>	The background color to use displaying selected text.
<b>state</b>	Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED
<b>wrap</b>	This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too long will be broken at any character.
<b>xscrollcommand</b>	To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.
<b>yscrollcommand</b>	To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.
<b>xview</b>	Query and change the horizontal position of the view.
<b>yview</b>	Query and change the vertical position of the view.

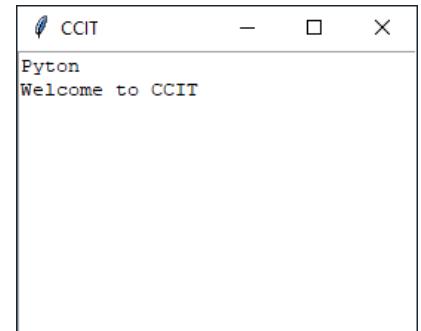
**Example:**

Program

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.title("CCIT")
        self.txt=Text(self)
        self.txt.pack()
        self.txt.insert(END,"Pyton\n")
        self.txt.insert(END,"Welcome to CCIT")
win=MyFrame()
win.mainloop()
```

Output



# Scrollbar

The Scrollbar class defines a new window and creates an instance of a scrollbar widget. Additional options, described below, may be specified in the method call or in the option database to configure aspects of the scrollbar such as its colors, orientation, and relief. The scrollbar method returns the identity of the new widget. At the time this command is invoked, the scrollbar's parent must exist.

A scrollbar is a widget that displays two arrows, one at each end of the scrollbar, and a slider in the middle portion of the scrollbar. It provides information about what is visible in an associated window that displays a document of some sort (such as a file being edited or a drawing). The position and size of the slider indicate which portion of the document is visible in the associated window..

## Syntax:

```
Scrollbar(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>get()</b>	It returns the two numbers a and b which represents the current position of the scrollbar.
<b>set(first, last)</b>	It is used to connect the scrollbar to the other widget w. The yscrollcommand or xscrollcommand of the other widget to this method.
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

Attribute	Description
<b>activebackground</b>	The background color of the widget when it has the focus.
<b>background ,bg</b>	background color

<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>command</b>	A procedure to be called whenever the scrollbar is moved.
<b>cursor</b>	The cursor that appears when the mouse is over the scrollbar.
<b>jump</b>	It is used to control the behavior of the scroll jump. If it set to 1, then the callback is called when the user releases the mouse button.
<b>orient</b>	It can be set to HORIZONTAL or VERTICAL depending upon the orientation of the scrollbar.
<b>takefocus</b>	We can tab the focus through this widget by default. We can set this option to 0 if we don't want this behavior.
<b>width</b>	It represents the width of the scrollbar.
<b>troughcolor</b>	It represents the color of the trough.

### Example:

#### Program

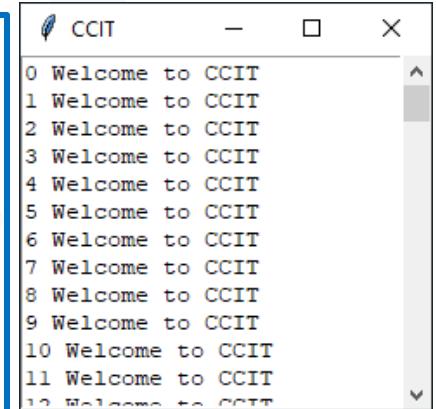
```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.title("CCIT")

        self.scroll=Scrollbar(self)
        self.scroll.pack(side=RIGHT,fill=Y)
        self.txt=Text(self)
        self.txt.configure(yscrollcommand=self.scroll.set)
        self.txt.pack()
        self.scroll.configure(command=self.txt.yview)

        for i in range(100):
            self.txt.insert(END,str(i)+" Welcome to CCIT\n")
win=MyFrame()
win.mainloop()
```

#### Output



# Frame

Python Tkinter Frame widget is used to organize the group of widgets. It acts like a container which can be used to hold the other widgets. The rectangular areas of the screen are used to organize the widgets to the python application.

The Frame class defines a new window and creates an instance of a frame widget. Additional options, described below, may be specified in the method call or in the option database to configure aspects of the frame such as its background color and relief. The frame command returns the path name of the new window. A frame is a simple widget. Its primary purpose is to act as a spacer or container for complex window layouts.

## Syntax:

```
Frame(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

Attribute	Description
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>height, width</b>	Sets height , Sets width
<b>font</b>	The default font for textual choices.
<b>padx</b>	The size of the internal padding added to the left and right of the text area. Default is one pixel.

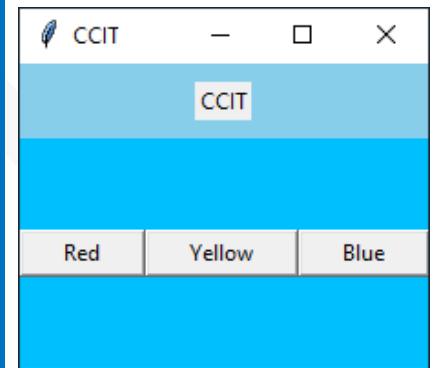
<b>pady</b>	The size of the internal padding added above and below the text area. Default is one pixel.
<b>highlightbackground</b>	Color of the focus highlight when the frame does not have focus..
<b>highlightcolor</b>	Color shown in the focus highlight when the frame has the focus.
<b>highlightthickness</b>	Thickness of the focus highlight.
<b>relief</b>	Border decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE.

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.title("CCIT")
        self.frm=Frame(self)
        self.frm.configure(background="skyblue",padx=10,
pady=10)
        self.frm.pack(fill='x')
        self.lbl=Label(self.frm,text="CCIT")
        self.lbl.pack()
        self.frm1=Frame(self)
        self.frm1.configure(background="deepskyblue",
pady=50)
        self.frm1.pack(fill='both',side='top')
        self.btn1=Button(self.frm1,text="Red",padx=50)
        self.btn1.grid(row=1,column=0)
        self.btn2=Button(self.frm1,text="Yellow", padx=50)
        self.btn2.grid(row=1,column=1)
        self.btn3=Button(self.frm1,text="Blue",padx=50)
        self.btn3.grid(row=1,column=2)

win=MyFrame()
win.mainloop()
```

**Output**

# LabelFrame

The LabelFrame widget is used to draw a border around its child widgets. We can also display the title for the LabelFrame widget. It acts like a container which can be used to group the number of interrelated widgets such as Radiobuttons. This widget is a variant of the Frame widget which has all the features of a frame. It also can display a label.

A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. This widget has the features of a frame plus the ability to display a label..

## Syntax:

```
LabelFrame(master=None, cnf={}, **kw)
```

## object methods

An instance of the class supplies the following methods

Method	Description
<b>pack( )</b>	To add widget into container(window).
<b>place( )</b>	To add widget into container(window).
<b>grid( )</b>	To add widget into container(window).
<b>config(**options)</b>	Modifies one or more widget options.

## configure method attributes

Attribute	Description
<b>text</b>	Specifies a string to be displayed inside the widget.
<b>background ,bg</b>	background color
<b>Borderwidth ,bd</b>	Size of border default is 2.
<b>height, width</b>	Sets height , Sets width
<b>font</b>	The default font for textual choices.
<b>padx</b>	The size of the internal padding added to the left and right of the text area. Default is one pixel.

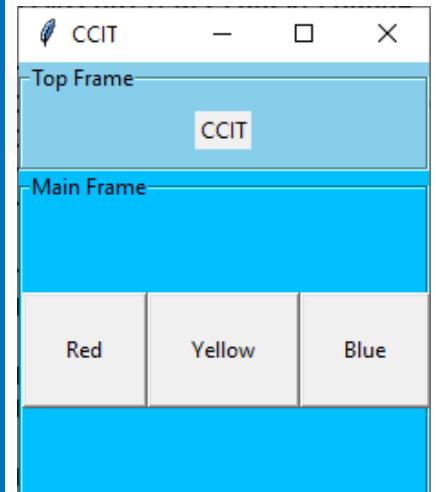
<b>pady</b>	The size of the internal padding added above and below the text area. Default is one pixel.
<b>highlightbackground</b>	Color of the focus highlight when the frame does not have focus..
<b>highlightcolor</b>	Color shown in the focus highlight when the frame has the focus.
<b>highlightthickness</b>	Thickness of the focus highlight.
<b>relief</b>	Border decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE.

**Example:****Program**

```
from tkinter import *

class MyFrame(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("400x300")
        self.title("CCIT")
        self.frm=LabelFrame(self,text="Top Frame")
        self.frm.configure(background="skyblue",padx=10,
pady=10)
        self.frm.pack(fill='x')
        self.lbl=Label(self.frm,text="CCIT")
        self.lbl.pack()
        self.frm1=LabelFrame(self,text="Main Frame")
        self.frm1.configure(background="deepskyblue",
pady=50)
        self.frm1.pack(fill='both',side='top')
        self.btn1=Button(self.frm1,text="Red",padx=20,
pady=20)
        self.btn1.grid(row=1,column=0)
        self.btn2=Button(self.frm1,text="Yellow",padx=20,
pady=20)
        self.btn2.grid(row=1,column=1)
        self.btn3=Button(self.frm1,text="Blue",padx=20,
pady=20)
        self.btn3.grid(row=1,column=2)

win=MyFrame()
win.mainloop()
```

**Output**

# Events and Binds

Events are notifications sent by the windowing system to the client code. They indicate that something has occurred or that the state of some controlled object has changed, either because of user input or because your code has made a request which causes the server to make a change.

In general, applications do not receive events automatically. However, you may not be aware of the events that have been requested by your programs indirectly, or the requests that widgets have made. For example, you may specify a command callback to be called when a button is pressed; the widget binds an activate event to the callback. It is also possible to request notification of an event that is normally handled elsewhere. Doing this allows your application to change the behavior of widgets and windows generally; this can be a good thing but it can also wreck the behavior of complex systems, so it needs to be used with care..

A Tkinter application runs most of its time inside an event loop, which is entered via the mainloop method. It waits for events to happen. Events can be key presses or mouse operations by the user. Tkinter provides a mechanism to let the programmer deal with events. For each widget, it's possible to bind Python functions and methods to an event.

Binding function is used to deal with the events. We can bind Python's Functions and methods to an event as well as we can bind these functions to any particular widget.

## Syntax:

```
Widget.bind(sequence, func, add= '')
```

**sequence** is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout's book for details).

**func** is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as callbacks.)

**add** is optional, either " or '+'. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a '+' means that this function is to be added to the list of functions bound to this event type.

## Events

Tkinter uses so-called event sequences for allowing the user to define which events, both specific and general, he or she wants to bind to handlers. It is the first argument "event" of the bind method. The event sequence is given as a string, using the following

### Syntax:

```
<modifier-type-detail>
```

The type field is the essential part of an event specifier, whereas the "modifier" and "detail" fields are not obligatory and are left out in many cases. They are used to provide additional information for the chosen "type". The event "type" describes the kind of event to be bound, e.g. actions like mouse clicks, key presses or the widget got the input focus.

Event	Description
<Button-1>	left mouse button click
<Button-2>	middle mouse button click
<Button-3>	right mouse button click
<Double-Button-1>	left mouse button double click
<Double-Button-2>	middle mouse button double click
<Double-Button-3>	right mouse button double click
<ButtonPress-1>	left mouse button Press
<ButtonPress-2>	middle mouse button Press
<ButtonPress-3>	right mouse button Press
<ButtonRelease-1>	left mouse button Release
<ButtonRelease-2>	middle mouse button Release
<ButtonRelease-3>	right mouse button Release
<B1-Motion>	mouse drag pressing left button
<B2-Motion>	mouse drag pressing middle button
<B3-Motion>	mouse drag pressing right button

<b>&lt;Enter&gt;</b>	The mouse pointer entered the widget.
<b>&lt;Leave&gt;</b>	The mouse pointer left the widget.
<b>&lt;FocusIn&gt;</b>	Keyboard focus was moved to this widget.
<b>&lt;FocusOut&gt;</b>	Keyboard focus was moved from this widget.
<b>&lt;Key&gt;</b>	The user pressed any key.
<b>&lt;key_name&gt;</b>	Key_name specify are alphabet keys numerical keys and special keys. The special keys are Cancel (the Break key), BackSpace, Tab, Return(the Enter key), Shift_L (any Shift key), Control_L (any Control key), Alt_L (any Alt key), Pause, Caps_Lock, Escape, Prior (Page Up), Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock, and Scroll_Lock.
<b>&lt;Control&gt;</b>	CONTROL key down.
<b>&lt;Return&gt;</b>	The user pressed the Enter key
<b>&lt;key1-key2&gt;</b>	The user pressed the key2, while holding the key1 pressed. You can use prefixes like Alt, Shift, and Control.

## Event Object

The event object is a standard Python object instance, with a number of attributes describing the event.

Attributes	Description
<b>widget</b>	The widget which generated this event. This is a valid Tkinter widget instance, not a name. This attribute is set for all events.
<b>type</b>	The event type.
<b>x, y</b>	The current mouse position, in pixels.
<b>x_root, y_root</b>	The current mouse position relative to the upper left corner of the screen, in pixels.
<b>char</b>	The character code (keyboard events only), as a string.
<b>keysym</b>	The key symbol (keyboard events only).
<b>keycode</b>	The key code (keyboard events only).
<b>num</b>	The button number (mouse button events only).

**Example:****Program**

```
from tkinter import *

def fun1(e):
    win.configure(bg="yellow")
def fun2(e):
    win.configure(bg="white")

win=Tk()
win.geometry("500x300")
win.configure(bg="white")
win.title("CCIT")

btn=Button(win,text="OK")
btn.place(relx=0.25,rely=0.25,relwidth=0.5,relheight=0.5)
btn.bind("<Enter>",fun1)
btn.bind("<Leave>",fun2)

win.mainloop()
```

**Output****Example:****Program**

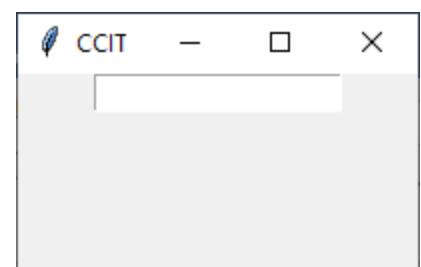
```
from tkinter import *

win = Tk()
win.geometry('200x100')
win.title('CCIT')

def showkey(event):
    print(event.keysym)

txt=Entry(win)
txt.pack()
txt.bind('<Key>', showkey)

win.mainloop()
```

**Output**

**Example:****Program**

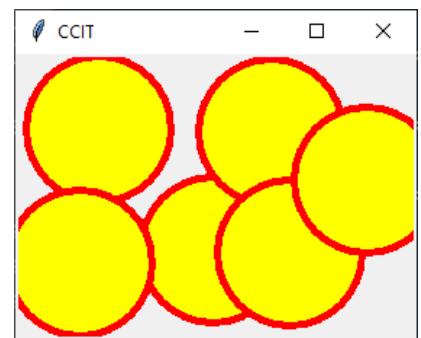
```
from tkinter import *

win = Tk()
win.title('CCIT')
win.geometry('500x400')

def show(e):
    can.create_oval([e.x-50,e.y-50,e.x+50,e.y+50],
fill="yellow",outline="red",width=5)

can=Canvas(win)
can.place(relwidth=1.0,relheight=1.0)
can.bind('<Button-1>', show)

win.mainloop()
```

**Output****Example:****Program**

```
from tkinter import *

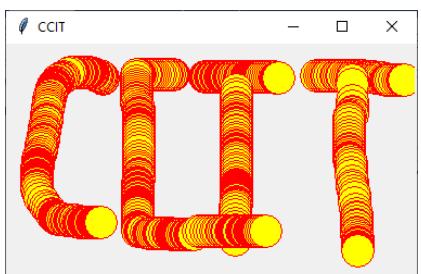
win = Tk()
win.title('CCIT')
win.geometry('500x400')

def show(e):
    can.create_oval([e.x-15,e.y-15,e.x+15,e.y+15],
fill="yellow",outline="red")

can=Canvas(win)
can.place(relwidth=1.0,relheight=1.0)

can.bind('<B1-Motion>', show)

win.mainloop()
```

**Output**

# Networking

---

---

# Networking

Computer Networking aims to study and analyze the communication process among various computing devices or computer systems that are linked, or networked together to exchange information and share resources. So, in order to perform networking you must have a network. It's like to make a telephone call you must have a telephone line. Isn't it obvious! So, there are basically four types of computer networks.

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

## Internet Protocol

The Internet Protocol (IP) is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries. Its routing function enables internetworking, and essentially establishes the Internet.

IP has the task of delivering packets from the source host to the destination host solely based on the IP addresses in the packet headers. For this purpose, IP defines packet structures that encapsulate the data to be delivered. It also defines addressing methods that are used to label the datagram with source and destination information.

Mainly, we will be dealing with two major protocols over the internet:

- User Datagram Protocol(UDP)
- Transmission Control Protocol(TCP)

### User Datagram Protocol(UDP)

- UDP transfers data without setting up a connection.
- It just sends datagram messages.
- In such type of communication a packet is created containing data and address of remote system to which data is to be send.
- Then this packet is send through the socket.
- At the destination system a socket must be ready to receive the packet otherwise the packet will be lost.

## TCP Protocol (Transmission Control Protocol )

- TCP is a connection-oriented protocol meaning it first sets up a connection to the receiver then sends the data in segments.
- It uses a Stream to send data to remote socket or to receive data from remote socket.

## Sockets

A socket is the end-point in a flow of communication between two programs or communication channels operating over a network. They are created using a set of programming requests called socket API (Application Programming Interface). Python's socket library offers classes for handling common transports as a generic interface.

These sockets are a combination of an IP address and a Port. A single device can have 'n' number of sockets based on the port number that is being used. Different ports are available for different types of protocols. Take a look at the following image for more about some of the common port numbers and the related protocols:

Protocol	Port	Function
HTTP	80	Web page
FTP	20	File transfers
SMTP	25	Sending email
Telnet	23	Command line

## socket Module

Python's socket module handles networking with the socket interface. There are minor differences between platforms, but the module hides most of them, making it relatively easy to write portable networking applications. The module defines exception class `socket.error`.

Function	Description
<code>gethostbyaddr(ip_address)</code>	Takes a string containing an IPv4 or IPv6 address and returns a three-item tuple of the form ( <code>hostname, aliaslist, ipaddrlist</code> ). <code>hostname</code> is the canonical name for the IP
<code>gethostbyname(hostname)</code>	Returns a string containing the IPv4 address associated with the given <code>hostname</code> . If called with an IP address, returns that address.
<code>getnameinfo(sock_addr, flags=0)</code>	Takes a socket address and returns a ( <code>host, port</code> ) pair. Without flags, <code>host</code> is an IP address and <code>port</code> is an int.

# Class socket

The socket class is the primary means of network communication in Python. A new socket is also created when a SOCK\_STREAM socket accepts a connection, each such socket being used to communicate with the relevant client.

## Syntax:

```
socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
```

Create a new socket using the given address family, socket type and protocol number. The address family should be AF\_INET (the default), AF\_INET6, AF\_UNIX, AF\_CAN, AF\_PACKET, or AF\_RDS. The socket type should be SOCK\_STREAM (the default), SOCK\_DGRAM, SOCK\_RAW or perhaps one of the other SOCK\_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is AF\_CAN the protocol should be one of CAN\_RAW, CAN\_BCM or CAN\_ISOTP.

If fileno is specified, the values for family, type, and proto are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit family, type, or proto arguments.

## object methods

An instance of the class supplies the following methods

Method	Description
<b>accept()</b>	Blocks until a client establishes a connection to s, which must have been bound to an address (with a call to s.bind) and set to listening (with a call to s.listen). Returns a new socket object, which can be used to communicate with the other endpoint of the connection.
<b>bind(address)</b>	Binds s to a specific address. The form of the address argument depends on the socket' s address family
<b>close()</b>	Marks the socket as closed. It does not necessarily close the connection immediately, depending on whether other references to the socket exist.
<b>connect(address)</b>	Connects to a remote socket at address.
<b>listen([backlog])</b>	Starts the socket listening for traffic on its associated endpoint. If given, the integer backlog argument determines how many unaccepted connections the operating system allows to queue up before starting to refuse connections.

<b>recv(bufsiz)</b>	Receive a maximum of bufsiz bytes of data on the socket. Returns the received data.
<b>recvfrom(bufsiz)</b>	Receive a maximum of bufsiz bytes of data from s.
<b>send(bytes)</b>	Send the given data bytes over the socket, which must already be connected to a remote endpoint.
<b>sendall(bytes)</b>	Send all the given data bytes over the socket, which must already be connected to a remote endpoint.
<b>sendto(bytes, address)</b>	Transmit the bytes (s must not be connected) to the given socket address.
<b>sendfile(file)</b>	Send the contents of file object file (which must be open in binary mode) to the connected endpoint.

## Server Example:

### Program

```
import socket
server=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.bind(('localhost',5555))
server.listen()

while True:
    print("waiting for client request...")
    cskt,caddr=server.accept()
    req=cskt.recv(1024)
    req=req.decode()
    print("request received :" +req)
    res="unknown req"
    if req=="pena":
        res="pepsi"
    if req=="khana":
        res="pizza"
    cskt.send(res.encode('utf-8'))
    print("response send...")
    cskt.close()

server.close()
```

### Output

```
waiting for client request...
request received :khana
response send...
waiting for client request...
```

## Client Example:

Program

```
import socket

req=input("Enter Request: ")
cskt=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
cskt.bind(('localhost',2222))
cskt.connect(('localhost',5555))
cskt.send(req.encode('utf-8'))
res=cskt.recv(1024)
print("Response is :",res.decode())
cskt.close()
```

Output

```
Enter Request: khana
Response is : pizza
```

# sys Module

---

---

# sys Module

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. The python sys module provides functions and variables which are used to manipulate different parts of the Python Runtime Environment. It lets us access system-specific parameters and functions.

<b>Attributes and method</b>	<b>Description</b>
<b>argv</b>	The list of command line arguments passed to a Python script. argv[0] is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the -c command line option to the interpreter, argv[0] is set to the string '-c'. If no script name was passed to the Python interpreter, argv[0] is the empty string.
<b>maxint</b>	The largest int in this version of Python (at least $2^{**}31-1$ ; that is, 2147483647).
<b>maxsize</b>	Maximum number of bytes in an object in this version of Python (at least $2^{**}31-1$ ; that is, 2147483647).
<b>modules</b>	A dictionary whose items are the names and module objects for all loaded modules.
<b>path</b>	A list of strings that specifies the directories and ZIP files that Python searches when looking for a module to load.
<b>platform</b>	A string that names the platform on which this program is running. Typical values are brief operating system names, such as 'darwin', 'linux2', and 'win32'.
<b>stdin</b>	This is the file-handle that a user program reads to get information from the user. We give input to the standard input (stdin).
<b>stdout</b>	The user program writes normal information to this file-handle. The output is returned via the Standard output (stdout).
<b>stderr</b>	The user program writes error information to this file-handle. Errors are returned via the Standard error (stderr).
<b>version</b>	A string that describes the Python version, build number and date, and Compiler used.
<b>exit()</b>	Exit from Python. This is implemented by raising the SystemExit exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level.

## Command Line Arguments

Python sys module stores the command line arguments into a list, we can access it using sys.argv. This is very useful and simple way to read command line arguments as String. Command line arguments send to a program can be retrieved by using a list sys.argv. sys.argv is the list of command-line arguments. len(sys.argv) is the number of command-line arguments. sys.argv[0] is the program ie. the script name. sys.argv[1] contains 1st argument and so on.

### Example:

Program

```
import sys
n=int(sys.argv[1])
f=1
for i in range(1,n+1):
    f=f*i
print("Factorial of ",n," is ",f)
```

Command prompt

```
C:/python>python demo.py 5
Factorial of 5 is 120
```

### Example:

Program

```
import sys
if len(sys.argv)==4:
    p=int(sys.argv[1])
    r=float(sys.argv[2])
    n=int(sys.argv[3])
    si=p*r*n/100
    print("Simple Interest is ",si)
else:
    print("usage: interest <P> <R> <N>")
```

Command prompt

```
C:/python>python demo.py 2000 12.45
2
Simple Interest is 498.0
```

## stdin, stdout, stderr

stdin, stdout, and stderr are predefined file-like objects that correspond to Python's standard input, output, and error streams. You can rebind stdout and stderr to file-like objects open for writing (objects that supply a write method accepting a string argument) to redirect the destination of output and error messages. You can rebind stdin to a file-like object open for reading (one that supplies a readline method returning a string).

### Example:

#### Program

```
import sys

sys.stdout.write("Enter Your Name\n")
nm=sys.stdin.readline()
sys.stdout.write(f"Hello, {nm}")
```

#### Output

```
Enter Your Name
Amit Jain
Hello, Amit Jain
```

### Example:

#### Program

```
import sys

sys.stdout.write(f"Enter radius\n")
val=sys.stdin.readline()
r=int(val)
if r>=0:
    a=3.14*r**2
    c=2*3.14*r
    sys.stdout.write(f"\n Area is {a}")
    sys.stdout.write(f"\n Circumference is {c}")
else:
    sys.stderr.write("\n radius cannot be negative")
```

#### Output

```
Enter radius
5
Area is 78.5
Circumference is 31.400000002
```

```
Enter radius
```

-6

**radius cannot be negative**

# math Module

---

---

# math Module

The Python math module is an important feature designed to deal with mathematical operations. It comes packaged with the standard Python release and has been there from the beginning. Most of the math module's functions are thin wrappers around the C platform's mathematical functions..

## Constant

constant	Description
<b>pi</b>	The mathematical constant $\pi = 3.141592\dots$ , to available precision..
<b>e</b>	The mathematical constant $e = 2.718281\dots$ , to available precision.
<b>tau</b>	The mathematical constant $\tau = 6.283185\dots$ , to available precision. Tau is a circle constant equal to $2\pi$ , the ratio of a circle's circumference to its radius.
<b>inf</b>	A floating-point positive infinity. (For negative infinity, use -math.inf.) Equivalent to the output of float('inf').
<b>nan</b>	A floating-point "not a number" (NaN) value. Equivalent to the output of float('nan').

## Number-theoretic and representation functions

function	Description
<b>ceil(x)</b>	Return the ceiling of x, the smallest integer greater than or equal to x.
<b>comb(n, k)</b>	Return the number of ways to choose k items from n items without repetition and without order.
<b>copysign(x,y)</b>	Return a float with the magnitude (absolute value) of x but the sign of y. On platforms that support signed zeros, copysign(1.0, -0.0) returns -1.0.
<b>fabs(x)</b>	Return the absolute value of x.
<b>factorial(x)</b>	Return x factorial as an integer.
<b>floor(x)</b>	Return the floor of x, the largest integer less than or equal to x.
<b>fmod(x, y)</b>	Return fmod(x, y), as defined by the platform C library. Note that the Python expression x % y may not return the same result.
<b>fsum(iterable)</b>	Return an accurate floating point sum of values in the iterable.

## Angular conversion

function	Description
<b>degrees(x)</b>	Convert angle x from radians to degrees.
<b>radians(x)</b>	Convert angle x from degrees to radians.

## Power and logarithmic functions

function	Description
<b>log(x,base)</b>	With one argument, return the natural logarithm of x (to base e). With two arguments, return the logarithm of x to the given base, calculated as $\log(x)/\log(\text{base})$ .
<b>log2(x)</b>	Return the base-2 logarithm of x. This is usually more accurate than $\log(x, 2)$ .
<b>log10(x)</b>	Return the base-10 logarithm of x. This is usually more accurate than $\log(x, 10)$ .
<b>pow(x, y)</b>	Returns the value of x to the power of y
<b>sqrt(x)</b>	Return the square root of x.

## Trigonometric functions

function	Description
<b>cos(x)</b>	Return the cosine of x radians.
<b>sin(x)</b>	Return the sine of x radians.
<b>tan(x)</b>	Return the tangent of x radians.

### Example:

Program

```
import math
r=5
a=math.pi*r**2
c=math.tau*r

print("Area is ",a)
print("Circumference is ",c)
```

Output

```
Area is 78.53981633974483
Circumference is
31.41592653589793
```

# random Module

# random Module

This module implements pseudo-random number generators for various distributions. For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

Python offers random module that can generate random numbers. These are pseudo-random number as the sequence of number generated depends on the seed. If the seeding value is same, the sequence will be the same.

## Functions for integers

Function	Description
<b>randrange(start, stop, step)</b>	The randrange() method returns a randomly selected element from the specified range. Returns a random integer from the range
<b>randint(a, b)</b>	The randint() method returns an integer number selected element from the specified range. Returns a random integer between a and b inclusive

## Functions for sequences

Function	Description
<b>choice(seq)</b>	Return a random element from the non-empty sequence seq. If seq is empty, raises IndexError. Return a random element from the non-empty sequence
<b>choices(population, weights=None, k=1)</b>	Return a k sized list of elements chosen from the population with replacement. If the population is empty, raises IndexError. If a weights sequence is specified, selections are made according to the relative weights.
<b>shuffle(seq)</b>	The shuffle() method takes a sequence (list, string, or tuple) and reorganize the order of the items.
<b>sample(population, k)</b>	Return a k length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

## Real-valued distributions

Function	Description
<code>random()</code>	Return the next random floating point number in the range [0.0, 1.0).
<code>uniform(a,b)</code>	Return a random floating point number N such that $a \leq N \leq b$ for $a \leq b$ and $b \leq N \leq a$ for $b < a$ .
<code>triangular(low, high, mode)</code>	Return a random floating point number N such that $low \leq N \leq high$ and with the specified mode between those bounds. The low and high bounds default to zero and one.

### Example:

Program

```
import random

val=random.randint(0,25)
print("Random value",val)
```

Output

Random value 21

### Example:

Program

```
import random

lst=['C','C++','Java','Python']
val=random.choice(lst)
print("Random choice",val)
```

Output

Random choice C++

# Built-in Functions

---

---

# Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

<b>Built-in Functions</b>		
<code>abs()</code>	<code>format()</code>	<code>oct()</code>
<code>all()</code>	<code>frozenset()</code>	<code>open()</code>
<code>any()</code>	<code>getattr()</code>	<code>ord()</code>
<code>ascii()</code>	<code>globals()</code>	<code>pow()</code>
<code>bin()</code>	<code>hasattr()</code>	<code>print()</code>
<code>bool()</code>	<code>hash()</code>	<code>property()</code>
<code>breakpoint()</code>	<code>help()</code>	<code>range()</code>
<code>bytearray()</code>	<code>hex()</code>	<code>repr()</code>
<code>bytes()</code>	<code>id()</code>	<code>reversed()</code>
<code>callable()</code>	<code>input()</code>	<code>round()</code>
<code>chr()</code>	<code>int()</code>	<code>set()</code>
<code>classmethod()</code>	<code>isinstance()</code>	<code>setattr()</code>
<code>compile()</code>	<code>issubclass()</code>	<code>slice()</code>
<code>complex()</code>	<code>iter()</code>	<code>sorted()</code>
<code>delattr()</code>	<code>len()</code>	<code>staticmethod()</code>
<code>dict()</code>	<code>list()</code>	<code>str()</code>
<code>dir()</code>	<code>locals()</code>	<code>sum()</code>
<code>divmod()</code>	<code>map()</code>	<code>super()</code>
<code>enumerate()</code>	<code>max()</code>	<code>tuple()</code>
<code>eval()</code>	<code>memoryview()</code>	<code>type()</code>
<code>exec()</code>	<code>min()</code>	<code>vars()</code>
<code>filter()</code>	<code>next()</code>	<code>zip()</code>
<code>float()</code>	<code>object()</code>	<code>__import__()</code>

# 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.

## Syntax:

```
abs(num)
```

## Parameters

The abs() method takes a single argument:

num - number whose absolute value is to be returned. The number can be:

- integer
- floating number
- complex number

## Return value

The abs() method returns the absolute value of the given number.

- For integers - integer absolute value is returned
- For floating numbers - floating absolute value is returned
- For complex numbers - magnitude of the number is returned

## Example:

### Program

```
val1=-52
print("Absolute value is",abs(val1))
val2=-15.33
print("Absolute value is",abs(val2))
```

### Output

```
Absolute value is 52
Absolute value is 15.33
```

# any function

The any() function returns True if any element of an iterable is True. If not, any() returns False..

## Syntax:

```
any(iterable)
```

## Parameters

The any() function takes an iterable (list, string, dictionary etc.) in Python.

## Return value

The any() function returns a boolean value:

- True if at least one element of an iterable is true
- False if all elements are false or if an iterable is empty

## Example:

### Program

```
l = [1, 3, 4, 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
```

# all function

The all() method returns True when all elements in the given iterable are true. If not, it returns False.

## Syntax:

```
all(iterable)
```

## Parameters

The all() function takes an iterable (list, string, dictionary etc.) in Python.

## Return value

The all() method returns:

- True - If all elements in an iterable are true
- False - If any element in an iterable is false

## Example:

### Program

```
l = [1, 3, 4, 5]
print(all(l))

l = [0, False]
print(all(l))

l = [0, False, 5]
print(all(l))

l = []
print(all(l))
```

### Output

```
True
False
False
True
```

# ascii function

The ascii() method returns a string containing a printable representation of an object. It escapes the non-ASCII characters in the string using \x, \u or \U escapes.

## Syntax:

```
ascii(object)
```

## Parameters

The ascii() method takes an object (strings, list etc).

## Return value

It returns a string containing printable representation of an object.

## Example:

### Program

```
Text = 'welcome to CCIT'  
  
print(ascii(Text))
```

### Output

```
welcome to CCIT
```

# bin function

The bin() method converts and returns the binary equivalent string of a given integer. If the parameter isn't an integer, it has to implement `__index__()` method to return an integer..

## Syntax:

```
bin(num)
```

## Parameters

The bin() method takes a single parameter:

num - an integer number whose binary equivalent is to be calculated.

## Return value

The bin() method returns the binary string equivalent to the given integer.

## Example:

### Program

```
number = 5  
print('binary of 5 is:', bin(number))
```

### Output

```
binary of 5 is: 0b101
```

# bool function

The `bool()` method converts a value to Boolean (True or False) using the standard truth testing procedure.

## Syntax:

```
bool(object)
```

## Parameters

Any object, like String, List, Number

## Return value

The `bool()` returns:

- False if the value is omitted or false
- True if the value is true

## Example:

### Program

```
number = 1
print('boolean value is:', bool(number))

number = 0
print('boolean value is:', bool(number))

val = ""
print('boolean value is:', bool(number))

val = "CCIT"
print('boolean value is:', bool(number))
```

### Output

```
boolean value is: True
boolean value is: False
boolean value is: False
boolean value is: True
```

# breakpoint function

Breakpoints are very convenient and can save you a lot of time. Instead of stepping through dozens of lines you're not interested in, simply create a breakpoint where you want to investigate.

## Syntax:

```
breakpoint()
```

## Parameters

It doesn't take any parameters.

## Return value

The breakpoint() returns no value.

## Example:

### Program

```
val = "Amravati"
print('value is:',val )

breakpoint()

val = "CCIT"
print('value is:',val)
```

### Output

```
value is: Amravati
>
c:\python\aaa.py(4)<module>()
-> val = "CCIT"
(Pdb)
```

# bytearray function

The bytearray() method returns a bytearray object which is an array of the given bytes.

## Syntax:

```
bytearray(source, encoding, errors)
```

## Parameters

The bytearray() takes three optional parameters:

- source (Optional) - source to initialize the array of bytes.
- encoding (Optional) - if source is a string, the encoding of the string.
- errors (Optional) - if source is a string, the action to take when the encoding conversion fails

## Return value

The bytearray() method returns an array of bytes of the given size and initialization values.

## Example:

### Program

```
val="Welcome to CCIT"  
data = bytearray(val, 'utf-8')  
print(data)
```

### Output

```
bytearray(b'Welcome to CCIT')
```

# bytes function

The bytes() method returns a immutable bytes object initialized with the given size and data.

## Syntax:

```
bytes(source, encoding, errors)
```

## Parameters

The bytes() takes three optional parameters:

- source (Optional) - source to initialize the array of bytes.
- encoding (Optional) - if source is a string, the encoding of the string.
- errors (Optional) - if source is a string, the action to take when the encoding conversion fails

## Return value

The bytes() method returns an array of bytes of the given size and initialization values.

## Example:

### Program

```
val="Welcome to CCIT"  
data = bytes(val, 'utf-8')  
print(data)
```

### Output

```
b'Welcome to CCIT'
```

# callable function

The callable() method returns True if the object passed appears callable. If not, it returns False.

## Syntax:

```
callable(object)
```

## Parameters

The callable() method takes a single argument object.

## Return value

The callable() method returns:

- True - if the object appears callable
- False - if the object is not callable.

## Example:

### Program

```
class msg:  
    def __call__(self):  
        print("Welcome to CCIT")  
  
m=msg()  
print(callable(msg))
```

### Output

```
True
```

# chr function

The chr() method returns a character (a string) from an integer (represents unicode code point of the character).

## Syntax:

```
chr(i)
```

## Parameters

The chr() method takes a single parameter, an integer i.

## Return value

The chr() returns a character (a string) whose Unicode code point is the integer i.

## Example:

### Program

```
print("Character value",chr(65))

print("Character value",chr(99))
```

### Output

```
A

c
```

# classmethod function

The classmethod() method returns a class method for the given function..

## Syntax:

```
classmethod(function)
```

## Parameters

The classmethod() method takes a single parameter:

- function - Function that needs to be converted into a class method

## Return value

The classmethod() method returns a class method for the given function.

## Example:

### Program

```
class Person:  
    age = 25  
  
def printAge(cls):  
    print('The age is:', cls.age)  
  
Person.printAge = classmethod(printAge)  
  
Person.printAge()
```

### Output

```
The age is: 25
```

# compile function

The compile() method returns a Python code object from the source (normal string, a byte string, or an AST object).

## Syntax:

```
compile(source, filename, mode)
```

## Parameters

source - a normal string, a byte string, or an AST object

filename - file from which the code was read. If it wasn't read from a file, you can give a name yourself

mode - Either exec or eval or single.

- eval - accepts only a single expression.
- exec - It can take a code block that has Python statements, class and functions and so on.
- single - if it consists of a single interactive statement

## Return value

The compile() method returns a Python code object.

## Example:

### Program

```
code="a=5; b=3;c=a+b;print('sum is',c)"  
obj=compile(code,__name__,'exec')  
exec(obj)
```

### Output

```
Sum is 8
```

# complex function

The `complex()` method returns a complex number when real and imaginary parts are provided, or it converts a string to a complex number.

## Syntax:

```
complex(real, imag)
```

## Parameters

In general, the `complex()` method takes two parameters:

- `real` - real part. If `real` is omitted, it defaults to 0.
- `imag` - imaginary part. If `imag` is omitted, it default to 0.

## Return value

As suggested by the name, the `complex()` method returns a complex number.

## Example:

### Program

```
c=complex(3,-5)  
print("Complex number is",c)
```

### Output

```
Complex number is (3-5j)
```

# delattr function

The delattr() deletes an attribute from the object (if the object allows it).

## Syntax:

```
delattr(object, name)
```

## Parameters

The delattr() takes two parameters:

- object - the object from which name attribute is to be removed
- name - a string which must be the name of the attribute to be removed from the object

## Return value

The delattr() doesn't return any value (returns None). It only removes an attribute (if object allows it).

## Example:

### Program

```
class Person:
    name="Amit"
    rollno=243

person=Person()
print("Name is ",person.name)
print("Rollno is ",person.rollno)

delattr(Person,'rollno')
print("Name is ",person.name)
print("Rollno is ",person.rollno)
```

### Output

```
Name is Amit
Rollno is 243
Name is Amit

AttributeError: 'Person'
object has no attribute
'rollno'
```

# dict function

The dict() constructor creates a dictionary in Python.

## Syntax:

```
dict(**kwarg)  
dict(mapping, **kwarg)  
dict(iterable, **kwarg)
```

## Parameters

A keyword argument is an argument preceded by an identifier (eg. name=). Hence, the keyword argument of the form kwarg=value is passed to the dict() constructor to create dictionaries.

## Return value

The dict() doesn't return any value (returns None).

## Example:

### Program

```
dis=dict([('Rollno','214'),('Name','Amit'))  
  
print(dis)
```

### Output

```
{'Rollno': '214', 'Name':  
'Amit'}
```

# dir function

The dir() method tries to return a list of valid attributes of the object.

## Syntax:

```
dir(object)
```

## Parameters

The dir() takes maximum of one object.

- object (optional) - dir() attempts to return all attributes of this object..

## Return value

The dir() tries to return a list of valid attributes of the object.

- If the object has \_\_dir\_\_() method, the method will be called and must return the list of attributes.
- If the object doesn't have \_\_dir\_\_() method, this method tries to find information from the \_\_dict\_\_ attribute (if defined), and from type object. In this case, the list returned from dir() may not be complete.

## Example:

### Program

```
val="CCIT"
print(dir(val))
```

### Output

```
[ '__add__', '__class__',
  '__contains__', '__doc__',
  '__eq__'...]
```

## Example:

### Program

```
class person:
    def __dir__(self):
        return ['Rollno', 'Name', 'Age']

p=person()
print(dir(p))
```

### Output

```
[ 'Age', 'Name', 'Rollno']
```

# divmod function

The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting of their quotient and remainder.

## Syntax:

```
divmod(x, y)
```

## Parameters

The divmod() takes two parameters:

- x - a non-complex number (numerator)
- y - a non-complex number (denominator)..

## Return value

The divmod() returns

(q, r) - a pair of numbers (a tuple) consisting of quotient q and remainder r

## Example:

### Program

```
print('divmod(5, 5) = ', divmod(5, 5))
```

### Output

```
divmod(5, 5) = (1, 0)
```

# enumerate function

The enumerate() method adds counter to an iterable and returns it (the enumerate object).

## Syntax:

```
enumerate(iterable, start=0)
```

## Parameters

The enumerate() method takes two parameters:

- iterable - a sequence, an iterator, or objects that supports iteration
- start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

## Return value

The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.

## Example:

### Program

```
lang=['C','C++','Java','Python']
enum=enumerate(lang)
print(list(enum))
```

### Output

```
[(0, 'C'), (1, 'C++'), (2, 'Java'), (3, 'Python')]
```

# eval function

The eval() method parses the expression passed to this method and runs python expression (code) within the program.

## Syntax:

```
eval(expression, globals=None, locals=None)
```

## Parameters

The eval() function takes three parameters:

- expression - the string parsed and evaluated as a Python expression
- globals (optional) - a dictionary
- locals (optional)- a mapping object. Dictionary is the standard and commonly used mapping type in Python..

## Return value

The eval() method returns the result evaluated from the expression..

## Example:

### Program

```
x=2  
print("Sum is ",eval('x+3'))
```

### Output

```
Sum is 5
```

# exec function

The exec() method executes the dynamically created program, which is either a string or a code object.

## Syntax:

```
exec(object, globals, locals)
```

## Parameters

The exec() takes three parameters:

- object - Either a string or a code object
- globals (optional) - a dictionary
- locals (optional)- a mapping object. Dictionary is the standard and commonly used mapping type in Python...

## Return value

The exec() doesn't return any value, it returns None.

## Example:

### Program

```
code="a=5; b=3;c=a+b;print('sum is',c)"  
exec(code)
```

### Output

```
Sum is 8
```

# filter function

The filter() method constructs an iterator from elements of an iterable for which a function returns true.

## Syntax:

```
exec(object, globals, locals)
```

## Parameters

The filter() method takes two parameters:

- function - function that tests if elements of an iterable returns true or false If None, the function defaults to Identity function - which returns false if any elements are false
- iterable - iterable which is to be filtered, could be sets, lists, tuples, or containers of any iterators..

## Return value

The filter() method returns an iterator that passed the function check for each element in the iterable.

## Example:

### Program

```
def even(n):
    if n%2==0:
        return True
    else:
        return False

num=[1,2,3,4,5,6,7,8,9,10]
result=filter(even,num)
for a in result:
    print(a)
```

### Output

```
2
4
6
8
10
```

# float function

The float() method returns a floating point number from a number or a string.

## Syntax:

```
float(x)
```

## Parameters

The float() method takes a single parameter:

- x (Optional) - number or string that needs to be converted to floating point number..

## Return value

The float() method returns equivalent floating point number if an argument is passed.

## Example:

### Program

```
a="2.45"  
print(float(a))
```

### Output

```
2.45
```

# format function

The built-in format() method returns a formatted representation of the given value controlled by the format specifier.

## Syntax:

```
format(value[, format_spec])
```

## Parameters

The format() function takes two parameters:

- value - value that needs to be formatted
- format\_spec - The specification on how the value should be formatted.

The format specifier could be in the format:

```
[[fill][align][sign][#][0][width][,][.precision][type]]  
where, the options are  
fill      ::=  any character  
align     ::=  "<" | ">" | "=" | "^"  
sign      ::=  "+" | "-" | "  
width     ::=  integer  
precision ::=  integer  
type      ::=  "b" | "c" | "d" | "f" | "F" | "o" | "s" | "x"
```

## Return value

The format() function returns a formatted representation of a given value specified by the format specifier.

### Example :

#### Program

```
a=25;  b=15.453521;  c="CCIT"  
print(format(a,"0>7"))  
print(format(b,".4f"))  
print(format(c,>7"))
```

#### Output

```
0000025  
15.4535  
CCIT
```

# frozenset function

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

## Syntax:

```
frozenset(iterable)
```

## Parameters

The frozenset() function takes a single parameter:

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

## Return value

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

## Example:

### Program

```
fset=frozenset(['C','C++','Java','Python'])  
print(fset)
```

### Output

```
frozenset({'Python', 'Java',  
'C++', 'C'})
```

# getattr function

The getattr() method returns the value of the named attribute of an object. If not found, it returns the default value provided to the function.

## Syntax:

```
getattr(object, name, default)
```

## Parameters

The getattr() method takes multiple parameters:

- object - object whose named attribute's value is to be returned
- name - string that contains the attribute's name
- default (Optional) - value that is returned when the named attribute is not found

## Return value

The getattr() method returns value of the named attribute of the given object.

## Example:

### Program

```
class Person:
    Rollno="214"
    Name="Amit Jain"
    Age="25"

person=Person()
print("Rollno:",getattr(person,"Rollno"))
print("Name:",getattr(person,"Name"))
print("Age:",getattr(person,"Age"))
```

### Output

```
Rollno: 214
Name: Amit Jain
Age: 25
```

# globals function

The `globals()` method returns the dictionary of the current global symbol table.

## Syntax:

```
globals()
```

## Parameters

The `globals()` method doesn't take any parameters.

## Return value

The `globals()` method returns the dictionary of the current global symbol table.

## Example:

### Program

```
print(globals())
```

### Output

```
{'__name__': '__main__',  
 '__doc__': None,  
 '__package__': None,  
 '__loader__': <class  
 '_frozen_importlib.BuiltinImp  
 orter'>, '__spec__': None,  
 '__annotations__': {},  
 '__builtins__': <module  
 'builtins' (built-in)>,  
 '__file__':  
 'C:/Python/demo.py'}
```

# hasattr function

The hasattr() method returns true if an object has the given named attribute and false if it does not.

## Syntax:

```
hasattr(object, name)
```

## Parameters

The hasattr() method takes two parameters:

- object - object whose named attribute is to be checked
- name - name of the attribute to be searched.

## Return value

The hasattr() method returns:

- True, if object has the given named attribute
- False, if object has no given named attribute.

## Example:

### Program

```
class Person:
    Rollno="214"
    Name="Amit Jain"

person=Person()
print("Rollno:",hasattr(person,"Rollno"))
print("Name:",hasattr(person,"Name"))
print("Age:",hasattr(person,"Age"))
```

### Output

```
Rollno: True
Name: True
Age: False
```

# hash function

The hash() method returns the hash value of an object if it has one.

## Syntax:

```
hash(object)
```

## Parameters

The hash() method takes a single parameter:

- object - the object whose hash value is to be returned (integer, string, float).

## Return value

The hash() method returns the hash value of an object if it has one.

## Example:

### Program

```
print('Hash for Python is:', hash('Python'))
```

### Output

```
Hash for Python is:  
2230730083538390373
```

# help function

The help() method calls the built-in Python help system.

## Syntax:

```
help(object)
```

## Parameters

The help() method takes maximum of one parameter.

- object (optional) - you want to generate the help of the given object).

## Return value

It return a help page is print.

## Example:

### Program

```
print(help(23))
```

### Output

```
Help on int object:

class int(object)
|   int([x]) -> integer
|   int(x, base=10) -> integer
|
|   Convert a number or string to an integer, or
|   return 0 if no arguments
|   are given. If x is a number, return x.__int__().
For floating point
|   numbers, this truncates towards zero.
|
|   If x is not a number or if base is given, then x
|   must be a string...
```

# hex function

The hex() function converts an integer number to the corresponding hexadecimal string.

## Syntax:

```
hex(x)
```

## Parameters

The hex() function takes a single argument.

- x - integer number (int object or it has to define \_\_index\_\_() method that returns an integer)

## Return value

The hex() function converts an integer to the corresponding hexadecimal number in string form and returns it..

## Example:

### Program

```
n1=214  
print(n1,'=',hex(n1))
```

### Output

```
214 = 0xd6
```

# id function

The id() function returns identity (unique integer) of an object.

## Syntax:

```
id(object)
```

## Parameters

The id() function takes a single parameter object.

## Return value

The id() function returns identity of the object. This is an integer which is unique for the given object and remains constant during its lifetime.

## Example:

### Program

```
class Person:  
    Rollno="258"  
    Name="Amit Jain"  
  
person=Person()  
print(id(person))
```

### Output

```
1310216203568
```

# input function

The input() method reads a line from input, converts into a string and returns it.

## Syntax:

```
input(prompt)
```

## Parameters

The input() method takes a single optional argument:

- prompt (Optional) - a string that is written to standard output (usually screen) without trailing newline.

## Return value

The input() method reads a line from input (usually user), converts the line into a string by removing the trailing newline, and returns it..

## Example:

### Program

```
data=input("Enter Your Name")
print("Hello,",data)
```

### Output

```
Enter Your Name Amit Jain
Hello, Amit Jain
```

# int function

The int() method returns an integer object from any number or string..

## Syntax:

```
int(x=0, base=10)
```

## Parameters

The int() method takes two arguments:

- x - Number or string to be converted to integer object. Default argument is zero.
- base - Base of the number in x. Can be 0 (code literal) or 2-36..

## Return value

The int() method returns:

- an integer object from the given number or string, treats default base as 10 (No parameters) returns 0.

## Example:

### Program

```
a=25  
b=35.614  
c='74'  
print(int(a))  
print(int(b))  
print(int(c))
```

### Output

```
25  
35  
74
```

# isinstance function

The `isinstance()` function checks if the object (first argument) is an instance or subclass of classinfo class (second argument)..

## Syntax:

```
isinstance(object, classinfo)
```

## Parameters

The `isinstance()` takes two parameters:

- object - object to be checked
- classinfo - class, type, or tuple of classes and types

## Return value

The `isinstance()` returns:

- True if the object is an instance or subclass of a class, or any element of the tuple False otherwise

## Example:

### Program

```
class Person:  
    Rollno=254  
    Name="Amit Jain"  
  
person=Person()  
print(isinstance(person,Person))
```

### Output

```
True
```

# issubclass function

The `issubclass()` function checks if the class argument (first argument) is a subclass of `classinfo` class (second argument).

## Syntax:

```
issubclass(class, classinfo)
```

## Parameters

The `issubclass()` takes two parameters:

- `class` - class to be checked
- `classinfo` - class, type, or tuple of classes and types

## Return value

The `issubclass()` returns:

- True if class is subclass of a class, or any element of the tuple False otherwise

## Example:

### Program

```
class Person:
    Name="Amit Jain"

class Employee(Person):
    empno=256
    age=26

    emp=Employee()
print(issubclass(Employee,Person))
print(issubclass(Person,Employee))
```

### Output

```
True
False
```

# iter function

The Python iter() function returns an iterator for the given object.

## Syntax:

```
iter(object, sentinel)
```

## Parameters

The iter() function takes two parameters:

- object - object whose iterator has to be created (can be sets, tuples, etc.)
- sentinel (optional) - special value that is used to represent the end of a sequence

## Return value

The iter() function returns an iterator object for the given object.

## Example:

### Program

```
lst=['C','C++','Java','Python']
i=iter(lst)

print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

### Output

```
C
C++
Java
Python
```

# len function

The len() function returns the number of items (length) in an object..

## Syntax:

```
len(s)
```

## Parameters

s - a sequence (string, bytes, tuple, list, or range) or a collection (dictionary, set or frozen set)

## Return value

The len() function returns the number of items of an object..

## Example :

### Program

```
lst=['C','C++','Java','Python']
print(len(lst))
```

### Output

```
4
```

# list function

The list() constructor returns a list in Python...

## Syntax:

```
list(iterable)
```

## Parameters

The list() constructor takes a single argument:

- iterable (optional) - an object that could be a sequence (string, tuples) or collection (set, dictionary) or any iterator object

## Return value

The list() constructor returns a list.

- If no parameters are passed, it returns an empty list
- If iterable is passed as a parameter, it creates a list consisting of iterable's items...

## Example:

### Program

```
lst=list(['C','C++','Java','Python'])  
print(lst)
```

### Output

```
['C', 'C++', 'Java',  
'Python']
```

# locals function

The locals() method updates and returns a dictionary of the current local symbol table

## Syntax:

```
locals()
```

## Parameters

The locals() method doesn't take any parameters.

## Return value

The locals() method updates and returns the dictionary associated with the current local symbol table.

## Example:

### Program

```
locals()
```

### Output

# map function

The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results.

## Syntax:

```
map(function, iterable)
```

## Parameters

function - map() passes each item of the iterable to this function. iterable - iterable which is to be mapped.

## Return value

The map() function applies a given to function to each item of an iterable and returns a list of the results.

## Example:

### Program

```
names=['amit', 'raj', 'mohan' ]  
itr = map( len , names )  
lst=list(itr)  
print(lst)
```

### Output

```
[ 4 , 3 , 5 ]
```

# max function

The Python max() function returns the largest item in an iterable. It can also be used to find the largest item between two or more parameters.

## Syntax:

```
max(iterable, *iterables, key, default)
```

## Parameters

- iterable - an iterable such as list, tuple, set, dictionary, etc.
- \*iterables (optional) - any number of iterables; can be more than one
- key (optional) - key function where the iterables are passed and comparison is performed based on its return value
- default (optional) - default value if the given iterable is empty

## Return value

The function returns the values of dictionaries. Based on the values (rather than the dictionary's keys), the key having the maximum value is returned.

If the items in an iterable are strings, the largest item (ordered alphabetically) is returned.

## Example:

### Program

```
lst=[2,65,7,45,35]  
print(max(lst))
```

### Output

```
65
```

# memoryview function

The memoryview() function returns a memory view object of the given argument. A memory view is a safe way to expose the buffer protocol in Python. It allows you to access the internal buffers of an object by creating a memory view object.

## Syntax:

```
memoryview(obj)
```

## Parameters

The memoryview() function takes a single parameter:

- obj - object whose internal data is to be exposed. obj must support the buffer protocol (bytes, bytearray)

## Return value

The memoryview() function returns a memory view object.

## Example :

### Program

```
data=b'CCIT'  
mv=memoryview(data)  
print(list(mv[0:4]))
```

### Output

```
[67, 67, 73, 84]
```

# min function

The Python min() function returns the smallest item in an iterable. It can also be used to find the smallest item between two or more parameters.

## Syntax:

```
min(iterable, *iterables, key, default)
```

## Parameters

- iterable - an iterable such as list, tuple, set, dictionary, etc.
- \*iterables (optional) - any number of iterables; can be more than one
- key (optional) - key function where the iterables are passed and comparison is performed based on its return value
- default (optional) - default value if the given iterable is empty

## Return value

In the case of dictionaries, min() returns the smallest key. Let's use the key parameter so that we can find the dictionary's key having the smallest value..

If the items in an iterable are strings, the smallest item (ordered alphabetically) is returned.

## Example:

### Program

```
lst=[22,65,7,45,35]  
print(min(lst))
```

### Output

```
7
```

# next function

The next() function returns the next item from the iterator.

## Syntax:

```
next(iterator, default)
```

## Parameters

- iterator - next() retrieves next item from the iterator
- default (optional) - this value is returned if the iterator is exhausted (there is no next item)

## Return value

The next() function returns the next item from the iterator.

If the iterator is exhausted, it returns the default value passed as an argument.

## Example:

### Program

```
lst=['C', 'C++', 'Java', 'Python']
i=iter(lst)

print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

### Output

```
C
C++
Java
Python
```

# object function

The object() function returns a featureless object which is a base for all classes..

## Syntax:

```
o = object()
```

## Parameters

The object() function doesn't accept any parameters.

## Return value

The object() function returns a featureless object.

## Example:

### Program

```
obj=object()  
print(dir(obj))
```

### Output

```
['__class__', '__delattr__',
 '__dir__', '__doc__',
 '__eq__', '__format__',
 '__ge__', '__getattribute__',
 '__gt__', '__hash__',
 '__init__'...]
```

# oct function

The oct() function takes an integer number and returns its octal representation.

## Syntax:

```
oct(x)
```

## Parameters

The oct() function takes a single parameter x.

This parameter could be:

- an integer number (binary, decimal or hexadecimal).

## Return value

The oct() function returns an octal string from the given integer number.

## Example:

### Program

```
a=32  
b=0x2D  
c=0b0101  
print(oct(a))  
print(oct(b))  
print(oct(c))
```

### Output

```
0o40  
0o55  
0o5
```

# ord function

The ord() function returns an integer representing the Unicode character.

## Syntax:

```
ord(ch)
```

## Parameters

The ord() function takes a single parameter:

- ch - a Unicode character

## Return value

The ord() function returns an integer representing the Unicode character..

## Example:

### Program

```
print(ord('A'))  
print(ord('c'))  
print(ord('$'))
```

### Output

```
65  
99  
36
```

# pow function

The pow() function returns the power of a number..

## Syntax:

```
pow(x, y, z)
```

## Parameters

The pow() function takes three parameters:

- x - a number, the base
- y - a number, the exponent
- z (optional) - a number, used for modulus

## Return value

The pow() function returns an integer value.

## Example:

### Program

```
print(pow(2,3))  
print(pow(3,5,10))
```

### Output

```
8  
3
```

# repr function

The repr() function returns a printable representation of the given object.

## Syntax:

```
repr(obj)
```

## Parameters

The repr() function takes a single parameter:

- obj - the object whose printable representation has to be returned

## Return value

The repr() function returns a printable representational string of the given object..

## Example:

### Program

```
val='Welcome to CCIT'  
print(repr(val))
```

### Output

```
'Welcome to CCIT'
```

# reversed function

The reversed() function returns the reversed iterator of the given sequence.

## Syntax:

```
reversed(seq)
```

## Parameters

The reversed() function takes a single parameter:

- seq - the sequence to be reversed

## Return value

The reversed() function returns an iterator that accesses the given sequence in the reverse order.

## Example:

### Program

```
val=['C','C++','Java','Python']
print(list(reversed(val)))
```

### Output

```
['Python', 'Java', 'C++',
'C']
```

# round function

The round() function returns a floating-point number rounded to the specified number of decimals.

## Syntax:

```
round(number, ndigits)
```

## Parameters

The round() function takes two parameters:

- number - the number to be rounded
- ndigits (optional) - number up to which the given number is rounded; defaults to 0

## Return value

If ndigits is not provided, round() returns the nearest integer to the given number.

If ndigits is given, round() returns the number rounded off to the ndigits digits.

## Example:

### Program

```
print(round(2.3))
print(round(45.5))
print(round(32.857))
```

### Output

```
2
46
33
```

# set function

The set() builtin creates a set in Python..

## Syntax:

```
set(iterable)
```

## Parameters

set() takes a single optional parameter:

- iterable (optional) - a sequence (string, tuple, etc.) or collection (set, dictionary, etc.) or an iterator object to be converted into a set.

## Return value

set() returns:

- an empty set if no parameters are passed
- a set constructed from the given iterable parameter

## Example:

### Program

```
print(set("Python"))
print(set(['C', 'C++', 'Python']))
```

### Output

```
{'h', 'y', 'P', 't', 'n',
'o'}
{'C++', 'Python', 'C'}
```

# setattr function

The setattr() function sets the value of the attribute of an object.

## Syntax:

```
setattr(object, name, value)
```

## Parameters

The setattr() function takes three parameters:

- object - object whose attribute has to be set
- name - attribute name
- value - value given to the attribute

## Return value

The setattr() method doesn't return anything; returns None.

## Example:

### Program

```
class Person:  
    Name="Amit Jain"  
  
person=Person()  
  
print(person.Name)  
setattr(person,"Name","Raj Kumar")  
print(person.Name)
```

### Output

```
Amit Jain  
Raj Kumar
```

# slice function

The slice() function returns a slice object that can be used to slice strings, lists, tuple etc.

## Syntax:

```
slice(start, stop, step)
```

## Parameters

slice() can take three parameters:

- start (optional) - Starting integer where the slicing of the object starts. Default to None if not provided.
- stop - Integer until which the slicing takes place. The slicing stops at index stop -1 (last element).
- step (optional) - Integer value which determines the increment between each index for slicing. Defaults to None if not provided.

## Return value

The slice() returns slice objects, let's you substring, sub-list, sub-tuple, etc.

## Example:

### Program

```
Name="Welcome to CCIT"

s1=slice(12)
s2=slice(2,6)
s3=slice(0,18,2)
print(Name[s1])
print(Name[s2])
print(Name[s3])
```

### Output

```
Welcome to C
lcom
Wloet CT
```

# sorted function

The sorted() function returns a sorted list from the items in an iterable.

## Syntax:

```
sorted(iterable, key=None, reverse=False)
```

## Parameters

sorted() can take a maximum of three parameters:

- iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.
- reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.
- key (Optional) - A function that serves as a key for the sort comparison. Defaults to None..

## Return value

The sorted() function sorts the elements of a given iterable in a specific order (either ascending or descending) and returns the sorted iterable as a list..

## Example:

### Program

```
lst=[26,5,48,23,84,75]  
  
print(sorted(lst))  
print(sorted(lst,reverse=True))
```

### Output

```
[5, 23, 26, 48, 75, 84]  
[84, 75, 48, 26, 23, 5]
```

# str function

The str() function returns the string version of the given object.

## Syntax:

```
str(object, encoding='utf-8', errors='strict')
```

## Parameters

The str() method takes three parameters:

- object - The object whose string representation is to be returned. If not provided, returns the empty string
- encoding - Encoding of the given object. Defaults of UTF-8 when not provided.
- errors - Response when decoding fails. Defaults to 'strict'.

## Return value

The str() method returns a string, which is considered an informal or nicely printable representation of the given object.

## Example:

### Program

```
a=25  
b=2.53  
c=b'CCIT'  
  
print(str(a))  
print(str(b))  
print(str(c, 'UTF-8'))
```

### Output

```
25  
2.53  
CCIT
```

# sum function

The sum() function adds the items of an iterable and returns the sum.

## Syntax:

```
sum(iterable, start)
```

## Parameters

- iterable - iterable (list, tuple, dict, etc). The items of the iterable should be numbers.
- start (optional) - this value is added to the sum of items of the iterable. The default value of start is 0 (if omitted).

## Return value

sum() returns the sum of start and items of the given iterable.

## Example:

### Program

```
print(sum([2,3,4,6,8,21]))  
print(sum([21,6.5,4.2,5]))  
print(sum([21,6,-5,-8]))
```

### Output

```
44  
36.7  
14
```

# tuple function

The tuple() builtin can be used to create tuples in Python..

## Syntax:

```
tuple(iterable)
```

## Parameters

- iterable (optional) - an iterable (list, range, etc.) or an iterator object

## Return value

tuple() returns a tuple is an immutable sequence type.

## Example:

### Program

```
t=tuple(['C','C++','Java','Python'])  
print(t)
```

### Output

```
('C', 'C++', 'Java',  
'Python')
```

# type function

The type() function either returns the type of the object or returns a new type object based on the arguments passed.

## Syntax:

```
type(object)
```

## Parameters

If a single object is passed to type(), the function returns its type.

## Return value

the function returns its object type..

## Example:

### Program

```
a=22  
b="CCIT"  
c=['Python','C']  
  
print(type(a))  
print(type(b))  
print(type(c))
```

### Output

```
<class 'int'>  
<class 'str'>  
<class 'list'>
```

# vars function

The vars() function returns the `__dict__` attribute of the given object..

## Syntax:

```
vars(object)
```

## Parameters

vars() takes a maximum of one parameter.

- object - can be module, class, instance, or any object having the `__dict__` attribute..

## Return value

vars() returns the `__dict__` attribute of the given object.

If the object passed to vars() doesn't have the `__dict__` attribute, it raises a `TypeError` exception...

## Example:

### Program

```
class Person:
    def __init__(self,a,b):
        self.Name=a
        self.Age=b

person=Person("Amit Jain",35)
print(vars(person))
```

### Output

```
{'Name': 'Amit Jain', 'Age': 35}
```

# zip function

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and return it.

## Syntax:

```
zip(*iterables)
```

## Parameters

- iterables - can be built-in iterables (like: list, string, dict), or user-defined iterables.

## Return value

The zip()function returns an iterator of tuples based on the iterable objects.

- If we do not pass any parameter, zip() returns an empty iterator
- If a single iterable is passed, zip() returns an iterator of tuples with each tuple having only one element.

## Example:

### Program

```
lst1=[1,2,3,4]
lst2=['C','C++','Java','Python']

result=zip(lst1,lst2)
print(list(result))
```

### Output

```
[(1, 'C'), (2, 'C++'), (3, 'Java'), (4, 'Python')]
```

# \_\_import\_\_ function

The \_\_import\_\_() is a function that is called by the import statement.

## Syntax:

```
__import__(name, globals=None, locals=None, fromlist=(), level=0)
```

## Parameters

- name - the name of the module you want to import
- globals and locals - determines how to interpret name
- fromlist - objects or submodules that should be imported by name
- level - specifies whether to use absolute or relative imports

## Return value

- This \_\_import\_\_() function is not necessary for everyday Python program. It is rarely used and often discouraged.
- This function can be used to change the semantics of the import statement as the statement calls this function. Instead, it is better to use import hooks..

## Example:

### Program

```
m=__import__('math',globals(),locals())
print(m.pow(5,2))
```

### Output

```
25.0
```