What is Python ?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What is Programming ?

We use programming language to communicate with computer.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
-

Installation of Python

- Open python.org
- Download latest version of python 3

What is IDEs ?

Integrated development environment

1. Software with usefull functionalists for developer
2. Productivity tools to help with software development
3. Has a graphical user interface
4. This help developer write build and test program

Why use IDEs ?

- Increase program productivity
- Source code editor
- File management

Popular IDE .

- PYCHARM
- XCODE
- Visual Studio code

Download Pycharm IDE .

- Go to pycharm download
- Download community

Modules

A modules is a file containing code written by somebody else , which can be imported and used in our programe

Type of module

Two type of module

1- Built in module – Pre install in python like os ,abc,arry,copy etc
2- External module – Need to install using pip like FLASK , TENSORFLOW etc

Install Modules

- Open window Powes
- Type PIP install flask (flask is module)

PIP

PIP is a package manager for python , you can use pip to install a module on your system.

Syntax

The syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be correctly structured statements or expressions in that language .

Print("hello world")

Variables

- Variables are containers for storing data values
- A Variable name can contain alphabet ,digit ,underscore
- Only start with an alphabet
- Cant start with digit
- No space allow inside a variable name

Comment

- Comments can be used to prevent execution when testing code.
    1. Single Line comment – #
    2. Multi line comment - Triple Quotes """ """

Function

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Operator in Python

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	**Name**	**Example**
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	**Example**	**Same As**	
=	Assign	x = 5	
+=	Add & assign	x = x + 3	
-=	Sub & ASSIGN	x = x - 3	
*=	Mutiply & assign	x = x * 3	
/=	Divide & assign	x = x / 3	
%=	Modulus & assign	x = x % 3	
//=	Floor division & assign	x = x // 3	
**=	Exponentiation & assign	x = x ** 3	
&=	x &= 3	x = x & 3	
|=	x |= 3	x = x | 3	
^=	x ^= 3	x = x ^ 3	
>>=	x >>= 3	x = x >> 3	
<<=	x <<= 3	x = x << 3	

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example	Try it
and	Returns True if both statements are true	x < 5 and x < 10	Try it »
or	Returns True if one of the statements is true	x < 5 or x < 4	Try it »
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)	

Python Identity Operators

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

Operator	Description	Example	Try it
is	Returns True if both variables are the same object	x is y	Try it »
is not	Returns True if both variables are not the same object	x is not y	

Python Membership Operators

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

Operator	**Description**	**Example**	**Try it**
in	Returns True if a sequence with the specified value is present in the object	x in y	Try it »
not in	Returns True if a sequence with the specified value is not present in the object	x not in y	Try it »

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Data type

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text type :	str
Numeric type :	int, float, complex
Sequence type :	list , tuple ,range
Mapping type :	dict
Set type :	set , frozenset
Boolean type :	bool
Binary type :	byte , bytearry , memoryview
None type :	nonetype

x=”hello world”	str
x= 20	int
x=20.5	float
x= 1j	complex
x = ["apple", "banana", "cherry"]	list
`x = ("apple", "banana", "cherry")`	`tuple`
`x=range(6)`	`range`
`x = {"name" : "John", "age" : 36}`	`dict`
x = {"apple", "banana", "cherry"}	set
`x = frozenset({"apple", "banana", "cherry"})`	`frozenset`
x = True	bool
`x = b"Hello"`	`bytes`
x = bytearray(5)	bytearray
`x = memoryview(bytes(5))`	`memoryview`
x=none	`none type`

Type() function use

Type() is used to find data type of a given variable in python

```
a=22
print(type(a))          <class “int”>

b="hey"
print(type(b))          <class “str”>

c=3.4
print(type(c))          <class “float”>

d=("hello","hey")
print(type(d))          <class “tuple”>
```

Input() function use

This function allows to take input from keyboard

```
Print(“enter yout number”)

a= input ()

print(“you enter this”,a)
```

Strings Data type

String is a sequence of character enclosed in QUOTES

`'hello'` is the same as `"hello"`

We can write string in 3 ways

- Single quotes – `'hello'`
- Double quote - `"hello"`
- Triple quotes (multiline string) -

```
"""hello
   World
I am amarth   """
```

```
print("Hello")
print('Hello')
```

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

```
a = "Hello, World!"
print(a[1])           e
```

Loop though a string -

we can loop through the characters in a string, with a `FOR` loop.

```
for x in "HEY":         H
  print(x)              E

                        Y
```

String length –

To get the length of a string, use the `len()` function

```
a = "Hello, World!"

print(len(a))          13

print(len("hey"))      3
```

Check string –

To check if a certain phrase or character is present in a string, we can use the keyword `in` .

```
a= "The best things in life are free!"

print("free" in a)      true
```

Use if keywod -

a = "The best things in life are free!"

if "free" in a:

print("Yes, 'free' is present.") **"Yes, 'free' is present."**

Use not in-

```
txt = "The best things in life are free!"
print("expensive" not in txt)                    true
```

Escape Characters

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

```
txt = "We are the so-called "Vikings" from the north."
```

```
#ERROR
```

To fix this problem, use the escape character `\"`:

```
txt = "We are the so-called \"Vikings\" from the north."
```

`#` We are the so-called "Vikings" from the north.

\v	vertical tab
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

Single quote

```
print("Who\'s this?")    # Who's this?
```

Backlash

want to print a single backslash?
We can't do it by just writing "", so we'll use "\\".

```
print("Interview\\Bit")      # interview\Bit
```

New line \n

```
print("Interview\nBit")                                  #interview
```
use “\n” here, which tells the interprtor   bit
print some characters in the new line separately.

Tab \t

```
print("Interview\tBit")        # interview   Bit
"""add tab space between words then this escape sequence will
```

Backspace \b

```
print("Interview \bBit")                  #interviewbit
```
#This escape sequence is used to remove the space between the words.

\r

```
print(r"Hello\nWorld")                    “hello world”
```
To ignore all the escape sequences in the string,
we have to make a string as a raw string using ‘r’ before the string

Index()

index() method returns the position at the first occurrence of the specified value.

String Silicing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
h	a	r	r	y		i	s		a		g		o	o	d		b	o	y
-20	-19	-18	-17	-16	-15	-14	-13	12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

a="harry is a good boy"

print(a[4])   #y
print(a[0:5]) #harry
print(a[0:75]) #harry is a good boy
print(a[0:])   #harry is a good boy
print(a[:])   #harry is a good boy
print(a[::])   #harry is a good boy
print(len(a))   #19   #FIND STRING LENGTH
print(a[4])   #y
print(a[0:5:2]) #hry   slicling with skiping
print(a[0:19:1])   #harry is a good boy
print(a[0::3])   #hriao y
print(a[::-1])   *#yob doog a si yrrah*
print(a[::-2])   *#ybdo iyrh*
print(a[-19:])   *#harry is a good boy*
print(a[-19:-1]) *#harry is a good bo*
print(a[-7:])   *#ood boy*
print(a[-19])   ***#h***

String Function

story = "once upon a time there was a youtuber named Harry who uploaded python course with notes Harry"

String Functions

```
print(len(story))                          #93
print(story.endswith("notes"))      #false
print(story.count("c"))                 #2
print(story.capitalize())        #Once upon a time there was a youtuber named harry who uploaded python course with notes harry
print(story.find("upon"))        #5

print(story.replace("Harry", "CodeWithHarry"))
#once upon a time there was a youtuber named CodeWithHarry who uploaded python course with notes CodeWithHarry
```

STRING FUNCTION

a = "i am amarth patel i am from Vidisha"
b= "a\tm\ta\tr\tt\tr\th"

#01
print(a.capitalize()) #Converts the first character to upper case   (I am amarth patel i am from vidisha)

#02
print(a.casefold()) #Converts string into lower case   (i am amarth patel i am from vidisha)

#03
print(a.count("i")) #Returns the number of times a specified value occurs in a string (4)

#04
print(a.encode())   #Returns an encoded version of the string (b'i am amarth patel i am from Vidisha')

#05
print(a.endswith("Vidisha")) #Returns true if the string ends with the specified value (true)

#06
print(a.endswith("from"))   #Returns true if the string ends with the specified value (false)

#07
print(b.expandtabs(4))   #Sets the tab size of the string (a   m   a   r   t   r   h)

#08
print(a.find("amarth"))   #Searches the string for a specified value and returns the position of where it was found (5)

#09
Formats specified values in a string

c= "my name is {name} , i am {age}"
print(c.format(name="amarth", age="23"))   #(my name is amarth , i am 23)

d="my name is {} , i am {}"
print(d.format("amarth","23"))   #(my name is amarth , i am 23)

e="my name is {0} , i am {1}"
print(e.format("amarth","23"))   #(my name is amarth , i am 23)

#10
print(a.index("amarth")) #searches the string for a specified value and returns the position of where it was found (5)

z= "iamamarthpatel1000"

#11
print(z.isalnum())   #Returns True if all characters in the string are alphanumeric means ,no space (true)
print(a.isalnum())   #Returns True if all characters in the string are alphanumeric means ,no space (false)

#12
print(z.isalpha()) #returns True if all characters in the string are in the alphabet ,no space (false)

#13
print(z.isascii())   #Returns True if all characters in the string are ascii characters (true)
print(a.isascii())   #Returns True if all characters in the string are ascii characters (true)

q= "\u0030"

#14
print(q.isdecimal()) #   Returns True if all characters in the string are decimals (true)

aa="123456789"

#15
print(aa.isdigit()) #Returns True if all characters in the string are digits   (true)

#16
The isidentifier() method returns True if the string is a valid identifier, otherwise False.
A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_)
A valid identifier cannot start with a number, or contain any spaces

a1 = "MyFolder"
b1= "Demo002"
c1 = "2bring"
d1 = "my demo"

```
print(a1.isidentifier())    #true
print(b1.isidentifier())    #true
print(c1.isidentifier())    #false
print(d1.isidentifier())    #false
```

#17
islower()   Returns True if all characters in the string are lower case

a = "hello world!"
b = "hello 123"
c = "mynameisPeter"

```
print(a.islower())  #(true)
print(b.islower())  #(true)
print(c.islower())  #(FALSE)
```

#18
isnumeric()   Returns True if all characters in the string are numeric
The isnumeric() method returns True if all the characters are numeric (0-9), otherwise False.
Exponents like ² and ¾ are also considered to be numeric values."-1" and "1.5" are NOT considered numeric values, because all the characters in the string must be numeric, and the - and the . are not.

a = "\u0030" #unicode for 0   (TRUE)
b = "\u00B2" #unicode for &sup2;(TRUE)
c = "10km2" (false)
d = "-1" (false)
e = "1.5" (false)
print(a.isnumeric())
print(b.isnumeric())
print(c.isnumeric())
print(d.isnumeric())
print(e.isnumeric())

#19
The istitle() method returns True if all words in a text start with a upper case letter,
AND the rest of the word are lower case letters, otherwise False.
Symbols and numbers are ignored.

a = "HELLO, AND WELCOME TO MY WORLD"
b = "Hello"
c = "22 Names"
d = "This Is %'!?"

```
print(a.istitle())          false
print(b.istitle())          true
print(c.istitle())          true
print(d.istitle())          true
```

#20
isprintable()   Returns True if all characters in the string are printable

#21
isspace()   Returns True if all characters in the string are whitespaces

#22
The isupper()   method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

```
a = "Hello World          false
b = "hello 123"            false
c = "MY NAME IS PETER"   true
print(a.isupper())
print(b.isupper())
print(c.isupper())
```

#23
maketrans() used for mapping table that can be used with the translate() method.

```
a="abc"                            #single string
dict={"a":"1","b":"2","c":"3"}
print(a.maketrans(dict))       #{97: '1', 98: '2', 99: '3'}

c="hello guys and welcome"                # single string
dict2={"a":"1","u":"2"}
print(c.maketrans(dict2))                #{97: '1', 117: '2'}

str="hello guys and welcome"
str1="abcde"
str2="12345"
print(str.maketrans(str1,str2))      #{97: 49, 98: 50, 99: 51, 100: 52, 101: 53}
print(chr(97))                    #a
print(chr(49))                    #1   interger representation

ss="hello guys$@ and welcome"
s1="abcd"
s2="1234"
s3="$@"                        #for delete
print(ss.maketrans(s1,s2,s3)) #{97: 49, 98: 50, 99: 51, 100: 52, 40: None, 36: None, 64: None}
print(chr(36))            #$
print(chr(64))            #@
print(chr(40))            #(
```

#24
The translate() method returns a string where some specified characters are replaced
#Use the maketrans() method to create a mapping table.
#If a character is not specified in the dictionary/table, the character will not be replaced.

```
string="i am amarth patel from vidisha"
dictionary= string.maketrans("a","9")
print(string.translate(dictionary))   #i 9m 9m9rth p9tel from vidish9

txt = "Hello Sam!"
mytable = txt.maketrans("S", "P")
print(txt.translate(mytable))        #Hello Pam!

txt1 = "Good night Sam!"
x = "mSa"
y = "eJo"
z = "odnght"                        # this part delete
mytable = txt1.maketrans(x, y, z)
print(txt1.translate(mytable))          #G i Joe!
```

#25
replace() method replaces a specified phrase with another specified phrase.

```
a="i am amarth"
print(a.replace("amarth","lucky"))  #i am lucky
```

#26
rfind() method finds the last occurrence of the specified value.

```
print(a.rfind("h"))  #10
```

#27
split() method splits a string into a list.

```
print(a.split()) #['i', 'am', 'amarth']
```

#28
splitlines() method splits a string into a list.

```
print(a.splitlines())    #['i am amarth']
```

#29
# startswith() method returns True if the string starts with the specified value,

```
print(a.startswith("i"))      #true
```

#30
swapcase() method returns a string where all the upper case letters are lower case and vice versa.
print(a.swapcase())   #I AM AMARTH   chanage krdeta hai small letter ko big main, big ko small main

#31
# title() method returns a string where the first character in every word is upper case
print(a.title())   #I Am Amarth

#32
upper() method returns a string where all characters are in upper case

#33
zfill() method adds zeros (0) at the beginning of the string, until it reaches the specified length.
z = "hello"
x= "welcome to the jungle"
c = "10.000"
print(z.zfill(10))   #00000hello
print(x.zfill(29))   #00000000welcome to the jungle
print(c.zfill(10))   #000010.000

List

Lists are used to store multiple items in a single variable. Like string , int, bool ,all.

Lists are created using square brackets [ ]

```
a = [1, 2 , 4, 56, 6]

print(a)    #[1 , 2, 4, 56, 6]


# Access using index using a[0], a[1], a[2]

print(a[2])   #4


# Change the value of list using

a[0] = 98

print(a) #[98, 2, 4, 56, 6]


# We can create a list with items of different types

c = [45, "Harry", False, 6.9]

print(c)   #[45, 'Harry', False, 6.9]
```

List method

#01

append() method appends(ADD) an element to the end of the list.

```
a = ["apple", "banana", "cherry"]

b = ["Ford", "BMW", "Volvo"]

a.append(b)

print(a) #['apple', 'banana', 'cherry', ['Ford', 'BMW', 'Volvo']]


c=[11,22,33,44]

a.append(c)

print(a) #['apple', 'banana', 'cherry', ['Ford', 'BMW', 'Volvo'], [11, 22, 33, 44]]


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

z.append(44)

print(z)  #[1, 2, 3, 4, 5, 44]
```

#02

clear() method removes all the elements from a list.

```
q=["amarth",2]

q.clear()

print(q)  #[]
```

#03

copy() method returns a copy of the specified list.

```
w=["apple", "banana", "cherry",2]

w.copy()

print(w) #['apple', 'banana', 'cherry', 2]
```

#04

count() method returns the number of elements with the specified value.

```
fruits = [1, 4, 2, 9, 7, 8, 9, 3, 1]

x = fruits.count(9)

print(x)    #2


fruits1 = ["apple", "banana", "cherry"]

x1 = fruits1.count("cherry")

print(x1)   #1
```

#05

extend() method adds the specified list elements to the end of the current list.

```
e = ['apple', 'banana', 'cherry',22]

r = ['Ford', 'BMW', 'Volvo']

e.extend(r)

print(e)   #['apple', 'banana', 'cherry',22, 'Ford', 'BMW', 'Volvo']
```

#06

index() method returns the position at the first occurrence of the specified value.

```
t = ['apple', 'banana', 'cherry',22]
y=t.index("apple")
print(y) #0

u=[11,22,33,44,55]
r=u.index(55)
print(r) #4
```

#07

insert() method inserts the specified value at the specified position.

```
aa = ['apple', 'banana', 'cherry',22]
aa.insert(1,"amarth")
print(aa)       #['apple', 'amarth', 'banana', 'cherry', 22]
```

#08

pop() method removes the element at the specified position.

```
ab = ['apple', 'banana', 'cherry',22]
ab.pop(1)
print(ab)        #['apple', 'cherry', 22]
```

#09

remove() method removes the first occurrence of the element with the specified value

```
ac=['apple', 'banana', 'cherry',22]

ac.remove(22)

print(ac) #['apple', 'banana', 'cherry']
```

#10

reverse() method reverses the sorting order of the elements.

```
ad=['apple', 'banana', 'cherry',22]

ad.reverse()

print(ad) #[22, 'cherry', 'banana', 'apple']
```

#11

sort() method sorts the list ascending by default.You can also make a function to decide the sorting criteria(s)

reverse = reverse=True will sort the list descending. Default is reverse=False

key   = A function to specify the sorting criteria(s)

```
ae=[1,4,32,5,6,76,767]

ae.sort()

print(ae)       #[1, 4, 5, 6, 32, 76, 767]


cars = ['Ford', 'BMW', 'Volvo']

cars.sort()

print(cars)     #['BMW', 'Ford', 'Volvo']


ad=[1,4,32,5,6,76,767]

ad.sort(reverse=True)

print(ad)    #[767, 76, 32, 6, 5, 4, 1]


ae=[1,4,32,5,6,76,767]

ae.sort(reverse=False)

print(ae)    #[1, 4, 5, 6, 32, 76, 767]
```

Tuple

A tuple is a immutable (cannot change) data type

Creating a tuple using () parenthesis bracket

```
a=()          - empty tuple
a =(1,)       - tuple with only one element write this way
a=(1,2,3)     - tuple with more than one element
```

```
b = (1,)
print(b)    #(1,)
```

two number swap

```
a= 1
b=8
a,b=b,a
print(a,b)   #8 1
```

Type of Tuple

#01

count() method returns the number of times a specified value appears in the tuple.

```
thistuple   =  (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)

x = thistuple.count(5)

print(x)      # 2
```

#02

The index() method finds the first occurrence of the specified value

```
thistuple2 = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)

x = thistuple2.index(8)

print(x)       #8
```

Set

sets is a collection of non repetition word .
means simply no repeat word

```
a=set()          #empty set
print(type(a))    #<class 'set'>

a = {1, 3, 4, 5, 1}
print(a)  #{1, 3, 4, 5}



# Creating an empty set
b = set()
print(type(b))


## Adding values to an empty set
b.add(4)
b.add(4)
b.add(5)
b.add(5   ) # Adding a value repeatedly does not changes a set
b.add((4, 5, 6))
print(b)      #{4, 5, (4, 5, 6)}
```

add()	Adds an element to the set
clear()	Removes all the elements from the set
copy( )	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update( )	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two or more sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with another set, or any other iterable

#01

```
# add() method adds an element to the set
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits) #{'banana', 'cherry', 'orange', 'apple'}
```

#02

The difference() method returns a set that contains the difference between two sets.
Meaning: The returned set contains items that exist only in the first set, and not in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.difference(y)
print(z) #{'banana', 'cherry'}
```

#03

The difference_update() method removes the items that exist in both sets
The difference_update() method is different from the difference() method, because the difference() method returns a new set, without the unwanted items, and the difference_update() method removes the unwanted items from the original set.

```
x1 = {"apple", "banana", "cherry"}
y1 = {"google", "microsoft", "apple"}
x1.difference_update(y1)
print(x1) #{'cherry', 'banana'}
```

#04

```
#The pop() method removes a random item from the set.
#This method returns the removed item.
x22=x1.pop()
print(x22) #banana
```

#05

```
# remove() method removes the specified element from the set.
xx={"aa","bb","cc"}
xx.remove("aa")
print(xx) #{'bb', 'cc'}
```

#06

```
#union() method returns a set that contains all items from the original set, and all
items from the specified set(s).
x6 = {"a", "b", "c"}
y6= {"f", "d", "a"}
z6 = {"c", "d", "e"}
result = x6.union(y6, z6)
print(result) #{'e', 'b', 'f', 'a', 'c', 'd'}

x66 = {"apple", "banana", "cherry"}
y66 = {"google", "microsoft", "apple"}
z66 = x.union(y66)
print(z66) #{'apple', 'google', 'cherry', 'microsoft', 'banana'}
```

#07

```
#isdisjoint() method returns True if none of the items are present in both sets,
otherwise it returns False.
x7 = {"apple", "banana", "cherry"}
y7 = {"google", "microsoft", "facebook"}
z7= x.isdisjoint(y7)
print(z7) #true
x77 = {"apple", "banana", "cherry"}
y77 = {"google", "microsoft", "cherry"}
z77= x77.isdisjoint(y77)
print(z77) #false
```

#08

```
#intersection() method returns a set that contains the similarity between two or
more sets.
z88=x77.intersection(y77)
print(z88) #{'cherry'}
```

DICTIONARY

key	value	KEY	VALUE	KEY	VALUE	KEY	VALUE
name	AMARTH	FROM	VIDISHA	AGE	23	education	graduate

DICTIONARY IS COLLECTION OF KEY AND VALUE

aaa = {"NAME":"AMARTH","FROM":"VIDISHA","AGE" : "23","education":"graduate"}

```
D={}
print(type(D))   #<class 'dict'>


D1= {"NAME":"AMARTH","FROM":"VIDISHA","AGE":"23","education":"graduate"}
print(D1["NAME"])      #AMARTH
print(D1["AGE"])       #23
print(D1) #{'NAME': 'AMARTH', 'FROM': 'VIDISHA', 'AGE': '23', 'education':
'graduate'}

D2={"harry":"burger","rohan":"fish","ram":{"meggi","paneer","chole"}}
D3={"harry":"burger","rohan":"fish","ram":{"a":"meggi","b":"paneer","c":"chole"}
}
print(D2["ram"]) #{'meggi', 'paneer', 'chole'}    dictionary ke andar dictionary
print(D3["ram"]["a"])   #meggi
```

#01
#clear() method removes all the elements from a dictionary.

```
D1= {"NAME":"AMARTH","FROM":"VIDISHA","AGE":"23","education":"graduate"}
D1.clear()
print(D1) #{}
```

```
#02
#Fromkeys() method returns a dictionary with the specified keys and the specified
value.
x = ('key1', 'key2', 'key3')
y = "HEY"
thisdict = dict.fromkeys(x, y)
print(thisdict) #{'key1': 'HEY', 'key2': 'HEY', 'key3': 'HEY'}

#03
#update() - hota hai

D4= {"NAME":"AMARTH","FROM":"VIDISHA","AGE":"23","education":"graduate"}
D4["vidisha"]="MP"
print(D4)   #{'NAME': 'AMARTH', 'FROM': 'VIDISHA', 'AGE': '23', 'education':
'graduate', 'vidisha': 'MP'}

D44=
{"NAME":"AMARTH","FROM":"VIDISHA","AGE":"23","education":"graduate"}
D44.update({"color": "White"})
print(D44) #{'NAME': 'AMARTH', 'FROM': 'VIDISHA', 'AGE': '23', 'education':
'graduate', 'color': 'White'}

#04
#get() method returns the value of the item with the specified key.
D5= {"NAME":"AMARTH","FROM":"VIDISHA","AGE":"23","education":"graduate"}
X=D5.get("NAME")
print(X)  #AMARTH

#05
#keys() method returns VALUE OF the keys of the dictionary
D6= {"NAME":"AMARTH","FROM":"VIDISHA","AGE":"23","education":"graduate"}
Z=D6.keys()
print(Z) #dict_keys(['NAME', 'FROM', 'AGE', 'education'])
```

```
#06
# value() return the value of value
O=D6.values()
print(O) #dict_values(['AMARTH', 'VIDISHA', '23', 'graduate'])

#07
#The pop() method removes the specified item from the dictionary.
D6.pop("NAME")
print(D6) #{'FROM': 'VIDISHA', 'AGE': '23', 'education': 'graduate'}
```

clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

If , elif , else

- Sometime we want to play game on our phone if day is sunday
  We go hiking if our parent allow.
- In python programming to we must be able to execute instruction on a condition being met
- if,elif,else are multiway decision taken by our programmer due to certain condition in our code .
- there can we any number 0f elif satatement
  last elif is executed only if all condition inside elif fail

This technique is known as Ternary Operators, or Conditional Expressions.

Python supports the usual logical conditions from mathematics:

Equals: a == b
Not Equals: a != b
Less than: a < b
Less than or equal to: a <= b
Greater than: a > b
Greater than or equal to: a >= b

Indentation - (whitespace at the beginning of a line print)

```
#if
a=10
if(a<11):
   print("yes")   #yes

a1 = 33
b1 = 200
if b1 > a1:
  print("ok")     #ok
```

#elif
elif is way of saying "if the previous conditions were not true, then try this condition".

```
a2 = 33
b2 = 33
if b2 > a2:
  print("b2 is greater than a2")
elif a2 == b2:
  print("a2 and b2 are equal")      #a2 and b2 are equal
```

#else
else catches anything which isn't caught by the preceding conditions

```
a3 = 200
b3= 33
if b3 > a3:
  print("b3 is greater than a3")
elif a3 == b3:
  print("a3 and b3 are equal")
else:
  print("a3 is greater than b3") #a3 is greater than b3
```

#short hand statement

```
if a3 > b3: print("a3 is greater than b3") #a3 is greater than b3

aa = 2000
bb = 330
print("A") if aa > bb else print("B")  #A

a1 = 330
b1 = 330
print("A") if a1 > b1 else print("=") if a1 == b1 else print("B")
```

# "and" and "or" keyword is a logical operator, and is used to combine conditional statements:

```
a = 200
b = 33
c = 500
if a > b or c > a:
  print("Both conditions are True")     #Both conditions are True
```

#You can have if statements inside if statements, this is called nested if statements.

```
x = 44
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

pass statement

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

```
a = 33
b = 200
if b > a:
  pass        #blankk
```

#practice

```
                    #take input from keyboard

v1=11
v2=22
v3=int(input())
if v3>v2:
  print("yes")
else:
  print("no")


p1=2
p2=4
p3=int(input())
if p3>p2:
  print("hello world")
elif p3==p2:
  print("hey world")
else:
  print("getlost")


# in , not in  keyward
list=[1,2,3]
print(2 in list)    #true
if 2 in list:
  print("ooyaa")   # ooyaa

if 5 not in list:
  print("hahah")  #hahah
```

Loop in Python FOR, WHILE

For Loop –

A for loop is used for iterating (repeat something) over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:                                    apple

  print(x)                                          banana

                                                    cherry
```

Looping Through a String

```
for x in "cat":                     c
  print(x)                          a
                                    t
```

Break Statement

With the break statement we can stop the loop before it has looped through all the items

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)

  if x == "banana":

    break                           # apple

                                      banana
```

Exit the loop when x is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

 if x == "banana":

  break

 print(x)                           #apple
```

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)          apple

                    cherry
```

range() Function

To loop through a set of code a specified number of times, we can use the `range()` function.

```
                            #0
for x in range(3):            1
  print(x)                    2
```

```
for x in range(2, 6):
  print(x)
```

```
for x in range(2, 30, 3):
  print(x)
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
a = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in a:
  for y in fruits:
    print(x, y)
```

output -

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

pass statement –

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

```
for x in [0, 1, 2]:
  pass
```

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Print a message once the condition is false:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Operator

Arithmetic operator

```
print("5 + 5 = ", 5+5 ) #addition   #25

print("5 - 5 = ", 5-5 )  #subtraction  #0

print("5 * 5 = ", 5*5 )   #multiplication #25

print("155 / 44 = ", 155/44 ) #division  #3.522727272727273

print("155 // 44 = ", 155//44 ) #floor division  decimal ni deta  #3

print("5 ** 2 = ", 5**2 )  #exponentiation     5 ki power 2    #25

print("5 % 3 = ", 5 % 3 ) #modulus remainder deta hai    #2
```

Assignment

```
x = 5
x+=7
print(x)   #12        add & assign

y=7
y-=5
print(y)    #2      sub & assign

z=5
z*=5
print(z)   #25     multi % assign
```

```
a=2
a/=2
print(a)   #1.0    division % assign



b=5
b**=5
print(b)   #3125   expo % assign

c=2
c//=2
print(c)   #1      floor division % assign

d=3
d%=28
print(d)    #3     modulus & assign
```

Comparision operator

```
i = 5
print( i == 4) #false       equal ==
print(i==5)     #true       equal ==
print(i != 4)   #true       not equal !=
print(i !=5)    #false      not equal !=
print(i > 4)     #true      greaterthan >
print(i < 6 )     #true     lessthan <
lessthan or equal to  <=
greatherthan or equal to >=
```

Logical operator

and operator

return true if both statement are True

x=5

print(x<7 and x >4)   # true 5 is smllerthn 7 and 5 is grtrthn 4 ,thts why its true

a= True

b= False

print(a and b) # FALSE

aa = False

BB= False

print(aa and bb) #False

or operator

retuen true if one statement are True

not operator

x = 5

print(not(x > 3 and x < 10))   #false

returns False because not is used to reverse the result

Identity operator

is used to compare the variable data is same or not

is

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)          #true
```

returns True because z is the same object as x

```
print(x is y)     #false
```

returns False because x is not the same object as y, even if they have the same content

```
print(x == y)        #true
```

to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y

is not

x = ["apple", "banana"]

y = ["apple", "banana"]

z = x

print(x is not z)   false

returns False because z is the same object as x

print(x is not y)   true

returns True because x is not the same object as y, even if they have the same content

print(x != y)   false

to demonstrate the difference betweeen "is not" and "!=": this comparison returns False because x is equal

Membership operator

used if a sequence is present in object

in

x = ["apple", "banana"]

print("banana" in x)

returns True because a sequence with the value "banana" is in the list

in not

x = ["apple", "banana"]

print("pineapple" not in x)

returns True because a sequence with the value "pineapple" is not in the list """

Bitwise operator

Short Hand “if”

```
a = int(input("enter a\n"))
b = int(input("enter b\n"))
print("a is bigger than b") if a>b else print("b is bigger than a ")
```

Function & Create

Function

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result

Creating a Function

In Python a function is defined using the **“def”** **keyword:**

Doc String

To open doc string file press ctrl and click on fuction

A Python docstring is a string used to document a Python module, class, function or method,

so programmers can understand what it does without having to read the details of the implementation.

user define function which is made by user

```
def function1():
    print("hello you are in function 1")
function1()                    #hello you are in function 1


def function2(a,b):
    print(a+b)
function2(1,4)                 #5


def function3(a,b):
    average=(a+b)/2
    print(average)
function3(5,7)                     #6.0
```

```
def function4(a,b):
    average=(a+b)/2
    return average              # value store in variable  aa main by typing
return
aa = function4(5,7)
print(aa)                       # 6.0



def function3(a,b):
    “ ""'THIS IS A FUNCTION WHICH WILL CALCULATE AVERAGE OF
TWO NUMBER"'"”      - THIS IS DOC STRING
     average=(a+b)/2
     print(average)
function3(5,7)
```

Arguments (args)

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses.

You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname).

When the function is called, we pass along a first name, which is used inside the function to print the full name

```
def my_function(fname):
  print(fname + " Refsnes")


my_function("Emil")                    #Emil Refsnes
my_function("Tobias")                   #Tobias Refnes
my_function("Linus")                   #Linus Refnes
```

Number of Arguments

By default, a function must be called with the correct number of arguments.

Meaning that if your function expects 2 arguments,you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments

```
def my_function(fname, lname):
  print(fname  + " " + lname)

my_function("Emil", "Refsnes")       # Emil Refsnes
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])


my_function("Emil", "Tobias", "Linus"                    #Linus
```

Arbitrary Arguments are often shortened to *args in Python documentations.

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)


my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus") # The youngest child is linus
```

The phrase Keyword Arguments are often shortened to kwargs in Python documentations

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function,

add two asterisk: ** before the parameter name in the function definition.

Example

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes") # His lat name is Refsnes
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")    # I am from Sweden
my_function("India")      #I am from India
my_function()             #I am from Norway
my_function("Brazil")    #I am from Brazil
```

Return Values

To let a function return a value, use the return statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))   #15
print(my_function(5))   #25
print(my_function(9))   #45
```

The Pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content,

put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass          #
```

TRY EXCEPT

try	block lets you test a block of code for errors.
Except	block lets you handle the error.
Else	block lets you execute code when there is no error.
Finally	block lets you execute code, regardless of the result of the try- and except blocks

```
print("Enter number 1")
a = input()
print("Enter number 2")
b = input()
try:
    print("the sum of these number is",int(a)+int(b))
except Exception as e :
    print(e)
    print("this line is very important")
```

#output

```
Enter number 1
e
Enter number 2
2
invalid literal for int() with base 10: 'e'
this line is very important
```

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

output

```
Variable x is not defined
```

#The try block will generate a NameError, because x is not defined:)

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

output

```
Hello
Nothing went wrong
```

The try block does not raise any errors, so the else block is executed:

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

output

```
Something went wrong
The 'try except' is finished
```

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1
if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

output

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise.py", line 4, in <module>
    raise Exception("Sorry, no numbers below zero")
Exception: Sorry, no numbers below zero
```

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"


if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

output

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise2.py", line 4, in <module>
    raise TypeError("Only integers are allowed")
TypeError: Only integers are allowed
```

File IO basics

non volatile memory - file

volatle memory - ram

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist (file ke end main add karna)

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Text mode defalut

"b" - Binary - Binary mode (e.g. images)

"+" - read and write (update file)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

open a file

first we made a file ap.txt

```
"amarth patel is king
he is smart
thik hai "
```

```
f = open("ap.txt")          #to open a file open() function use
c = f.read()                #to read a file read() fuunction use
print(c)
f.close()                   #to close a file close() function use
```

output

```
amarth patel is king
he is smart
thik hai
```

```
f = open("ap.txt","rt")        #rt text mode

c = f.read()

print(c)
```

output

```
amarth patel is king

he is smart

thik hai

thank you
```

```
f = open("ap.txt","rb")         #rb   binary mode

c = f.read()

print(c)               #b'amarth patel is king\r\nhe is smart\r\nthik hai\r\nthank
you\r\n'
```

Read Only Parts of the File

by default the **read()** method returns the whole text, but you can also specify how many characters you want to return:

```
f = open("ap.txt","r")

c = f.read(6)

print(c)            #amarth
```

Read Lines

You can return one line by using the readline() method:

```
f = open("ap.txt","rt")
print(f.readline())
print(f.readline())
```

output

```
amarth patel is king
he is smart
```

By looping through the lines of the file, you can read the whole file, line by line:

```
f = open("ap.txt")
for x in f:
    print(x)
```

output

```
he is smart
thik hai
thank you
```

```
f=open("ap.txt")
for x in f:
    print(x,end="")
```

output

```
amarth patel is king

he is smart

thik hai

thank you
```

```
f = open("ap.txt")

c=f.readlines()

print(c) #['amarth patel is king\n', 'he is smart\n', 'thik hai\n', 'thank you\n']
#list
```

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content , creat if file is not exist

```
f=open("ap.txt","w")                    #ap.txt file create hogi
f.write("hey i am amarth")
f.close()
```
output

```
hey i am amarth
```

```
f=open("ap.txt","")                     #"a" appand , line add in end
f.write("\nok i am amarth")
f.close()
```
output

```
hey i am amarth
ok i am amarth
```

```
f=open("ap.txt","w")
a = f.write("hey i am amarth")
print(a)                        #15
f.close()
```

handle read and write both

```
f=open("ap.txt","r+")
print(f.read())
f.write("thank you")
f.close()
```

output

```
hey i am amarth &
&
hey i am amarththank you
```

ap.txt file main add hoga

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, then delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the os.rmdir() method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove empty folders. """

Tell() , seek()

```
f=open("amarth","w")
f.write("hey i am amarth\niam from vidisha\ni am a student")
f.close()
```

"amarth" file is create

in file -

hey i am amarth

iam from vidisha

i am a student

```
f=open("amarth")
print(f.tell())                #0
print(f.readline())            #hey i am amarth
print(f.tell())                 #17
print(f.readline())              #iam from vidisha
print(f.tell())                        #35
print(f.readline())             #i am a student
```

```
f=open("amarth")
f.seek(9)
print(f.readline())                          #amarth  seek() read karta hai , 9 se
read karega
f.close()
```

with block to open file

using **"with"** to open a file

file- amarth

hey i am amarth

iam from vidisha

i am a student

```
with open ("amarth") as f:
  a =  f.readline()
  print(a)                    #hey i am amarth
```

file-ap.txt

hey i am amarth

thank you

i am good

```
with open("ap.txt") as f:
    print(f.read(3))              #hey
```

Variable in detail , global variable

Variables

Variables are containers for storing data values.

Multiple Variables

Python allows you to assign values to multiple variables in one line:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)      #Orange
print(y)      #Banana
print(z)      #Cherry
```

Note: Make sure the number of variables matches the number of values, or else you will get an error.

One Value to Multiple Variables

```
x = y = z = "Orange"
print(x)   #Orange
print(y)   #Orange
print(z)   #Orange
```

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

Example

Unpack a list:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)    apple
print(y)    banana
print(z)   cherry
```

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():

  print("Python is " + x)

myfunc()  # Python is awesome
```

If you create a variable with the same name inside a function,

this variable will be local, and can only be used inside the function.

The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"      #global variable

def myfunc():      #local variable

  x = "fantastic"

  print("Python is " + x)

myfunc()

print("Python is " + x)
```

output

```
Python is fantastic

Python is awesome
```

```
# example
a = 10
def myfun(n):
    print(n+"i write this")
myfun("amarth ")                         # amarth i write this



a = 10                         #global variable
def myfun(n):
    a=5                    #local variable
    m=3                #local variable
    print(a,m)
    print(n+"i write this")
myfun("s")
```

output

```
5 3
si write this
```

Want to change global variable value

```
l=10
def ok(n):
    m=8
    global l                         #global keyword
    l = l +12
    print(l,m)
ok(4)                    output-      22 8

def harry():
    x=20
    def rohan():
        global x
        x=12
    print("b c r rohan", x)
    rohan()
    print("a c rohan", x)
harry()
```

output

```
vb c r rohan 20
a c rohan 20
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.
Recursion is a common mathematical and programming concept.

It means that a function calls itself.
This has the benefit of meaning that you can loop through data to reach a result.

means function ke andar function use karna

fACTORIal formula n! =n * n-1 * n-2 * n-3.......1

```
 def factorial(n):                                    #function
    """
      :param n: Interger
      :return: n*n-1*n-2*n-3.......1
     """

   if n==1:
         return 1
    else:
        return n * factorial(n-1)
number=int(input("Enter the number\n"))
print(factorial(number))
```

output

```
input 5
 120
 5*factorial(4)
 5*4 * factorial(3)
 5*4*3 * factorial(2)
 5*4*3*2*factorial(1)
 5*4*3*2*1*=120
```

quiz

fibonacci sequence -
0 1 1 2 3 5 8 13 ....

```
def a(n):
    if n == 1:
        return 0
    elif n==2:
        return 1
    else:
        return a(n-1)+ a(n-2)
nn=int(input("entr your number"))
print(a(nn))
```

Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

lambda function or create function are same

lambda function is used in , to create one line function:

```
def minus(x,y):

    return x-y

print(minus(9,5))                #4
```

```
minus = lambda x,y:x-y
print(minus(9,5))                  #4

x = lambda a : a + 10
print(x(5))                        #15

x = lambda a, b : a * b
print(x(5, 6))                  #30

x = lambda a, b, c : a + b + c
print(x(5, 6, 2))                  #13



def myfunc(n):
  return lambda a : a * n
mytripler = myfunc(3)
print(mytripler(11))                  #33

def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))                  #22
print(mytripler(11))                #33
```

Modules

A module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension .py

Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the import statement:

Example

Import the module named mymodule, and call the greeting function:

```
import mymodule
mymodule.greeting("Jonathan")          #Hello, Jonathan
```

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule
a = mymodule.person1["age"]
print(a)                    #36
```

Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

Example

Create an alias for mymodule called mx:

```
import mymodule as mx
a = mx.person1["age"]
print(a)                    #36
```

Import From Module

You can choose to import only parts from a module, by using the from keyword.

Example

The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1
print (person1["age"])                    #36
```

Random Module

F String

f-Strings: A New and Improved Way to Format Strings in Python

f = F means FAST

```
name = "Eric"
age = 74
a= f"Hello, {name}. You are {age}."
print(a)
```

OUTPUT - 'Hello, Eric. You are 74.'

It would also be valid to use a capital letter F:

```
 F"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.
```

```
a= "amarth"
b=7
c=F"THIS IS {a} my NUMBER IS {b}"
print(c)                    #THIS IS amarth my NUMBER IS 7
```

```
 import math
a= "amarth"
b=7
c=F"THIS IS {a} my NUMBER IS {b} {math.cos(7)} "
print(c)                        #THIS IS amarth my NUMBER IS 7
0.7539022543433046
```

```
a= "amarth"
b=7
c=F"THIS IS {a} my NUMBER IS {b} {5*8} "
print(c)                             #THIS IS amarth my NUMBER IS 7 40
```

String formatting

The **format()** method formats the specified value(s) and insert them inside the string's placeholder.

The **placeholder** is defined using curly brackets: {}

The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}.

Syntax

string.format(value1, value2...)

Parameter Values

value1, value2... Required. One or more values that should be formatted and inserted in the string.

The values are either a list of values separated by commas, a key=value list, or a combination of both.

```
txt1 = "My name is {fname}, I'm {age}".format(fname = "John", age = 36)
txt2 = "My name is {0}, I'm {1}".format("John",36)
txt3 = "My name is {}, I'm {}".format("John",36)
```

output-

```
My name is John, I'm 36

My name is John, I'm 36

My name is John, I'm 36
```

:<	Left aligns the result (within the available space)
:>	Right aligns the result (within the available space)
:^	Center aligns the result (within the available space)
:=	Places the sign to the left most position
:+	Use a plus sign to indicate if the result is positive or negative
:-	Use a minus sign for negative values only
:	Use a space to insert an extra space before positive numbers
:,	Use a comma as a thousand separator
:_	Use a underscore as a thousand separator
:b	Binary format
:c	Converts the value into the corresponding unicode character
:d	Decimal format
:e	Scientific format, with a lower case e
:E	Scientific format, with an upper case E
:f	Fix point number format
:F	Fix point number format, in uppercase format (show inf and nan as INF and NAN)
:g	General format
:G	General format (using a upper case E for scientific notations)
:o	Octal format
:x	Hex format, lower case
:X	Hex format, upper case
:n	Number format
:%	Percentage format

Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

```
output-   The price is 49 dollars
```

Multiple Values

If you want to use more values, just add more values to the format() method:

```
print(txt.format(price, itemno, count))
```

And add more placeholders:

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

```
output -  I want 3 pieces of item number 567 for 2.00 dollars.
```

You can use index numbers (a number inside the curly brackets `{0}`) to be sure the values are placed in the correct placeholders:

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

```
output-   I want 3 pieces of item number 567 for 49.00 dollars.
```

```
age = 36
name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

```
output-  His name is John. John is 36 years old.
```

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

```
output-    I have a Ford, it is a Mustang.
```

Enumerate function

The enumerate() method adds a counter to an iterable and returns it (the enumerate object).

```
languages = ['Python', 'Java', 'JavaScript']

enumerate_prime = enumerate(languages)
# convert enumerate object to list
print(list(enumerate_prime))
```

Output : [(0, 'Python'), (1, 'Java'), (2, 'JavaScript')]

Syntax of enumerate()

The syntax of enumerate() is:

enumerate(iterable, start=0)

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.

You can convert enumerate objects to list and tuple using list() and tuple() method respectively.

Example 1: How enumerate() works in Python?

```
grocery = ['bread', 'milk', 'butter']

enumerateGrocery = enumerate(grocery)

print(type(enumerateGrocery))


# converting to list

print(list(enumerateGrocery))


# changing the default counter

enumerateGrocery = enumerate(grocery, 10)

print(list(enumerateGrocery))
```

Output

```
<class 'enumerate'>

[(0, 'bread'), (1, 'milk'), (2, 'butter')]

[(10, 'bread'), (11, 'milk'), (12, 'butter')]
```

Example 2: Looping Over an Enumerate object

```
grocery = ['bread', 'milk', 'butter']
for item in enumerate(grocery):
  print(item)

for count, item in enumerate(grocery):
  print(count, item)

for count, item in enumerate(grocery, 100):
  print(count, item)
```

Output

```
(0, 'bread')
(1, 'milk')
(2, 'butter')

0 bread
1 milk
2 butter
100 bread
101 milk
102 butter
```

If__name__==’__main__’

is used to import only function not contant .

example -

first we made tutmain1 file and tutmain2 file .

than we made function in tutmain1 file and write code and execute code .

but we only want to use function , not content(code)

we use **if__name__==__main__**

#example -

tutmain1.py file

```
def printhar(string):
    return f"ye string harry ko dede {string}"

def add(num1,num2):
    return num1+num2+5

print(printhar("harry1"))
o = add(4,6)                    #ye string harry ko dede harry1
print(o)                    #15
```

tutmain2 file

```
import tutmain1
print(tutmain1.add(5,3))
```

output_

```
ye string harry ko dede harry1
15
13
```

In tutmain2 , the content of tutmain1 also execute . to prevent this we use **if__name__==_main__**

the right way -

tutmain1 file -

```
def printhar(string):
    return f"ye string harry ko dede {string}"


def add(num1,num2):
    return num1+num2+5


print("and the name is", __name__)
if __name__ == '__main__':        #type main and enter
   print(printhar("harry1"))
   o = add(4,6)
   print(o)
```

output-

```
and the name is __main__
ye string harry ko dede harry1
15
```

tutmain2 file

```
import tutmain1
print(tutmain1.add(5,3))
```

output-

```
and the name is tutmain1
13
```

Join()

```
a= ["amarth","rahul","shreya","mahak","kat","virat"]
b= ",".join(a)
c=" and ".join(a)
d="-".join(a)
print(b)
print(c)
print(d)
```

output -

```
amarth,rahul,shreya,mahak,kat,virat
amarth and rahul and shreya and mahak and kat and virat
amarth-rahul-shreya-mahak-kat-virat
```

map()

The **map() function** executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax

map(function, iterables)

Parameter Values

Parameter	Description
function	Required. The function to execute for each item
iterable	Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

```
def myfunc(n):
  return len(n)

x = map(myfunc, ('apple', 'banana', 'cherry'))
```

print (list(x))

output -

```
<map object at 0x056D44F0>

[5, 6, 6]
```

```
def myfunc(a, b):
  return a + b
x = map(myfunc, ('apple', 'banana', 'cherry'),
('orange', 'lemon', 'pineapple'))

print(list(x))
```

output -

```
<map object at 0x034244F0>

['appleorange', 'bananalemon', 'cherrypineapple']
```

```
def seq(a):
   return a*a
numbers = [2,3,5,4,8,7,9,5,4,5]
square=list(map(seq,numbers))
print(square)                                        # [4, 9, 25, 16, 64, 49, 81, 25, 16, 25]


numbers = [2,3,5,4,8,7,9,5,4,5]
square=list(map(lambda x:x*x,numbers))          # [4, 9, 25, 16, 64, 49, 81, 25, 16, 25]
```

filter ()

The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.

Syntax

Filter (function, iterable)

Parameter Values

Parameter	Description
function	A Function to be run for each item in the iterable
iterable	The iterable to be filtered

```
k=[4,7,8,9,5,5,5,5,5,8,4,12,56,444,62,6,65,5]
def a(num):
   return num>5
b = list(filter(a,k))
print(b)
```

output-

```
[7, 8, 9, 8, 12, 56, 444, 62, 6, 65]
```

```
ages = [5, 12, 17, 18, 24, 32]
def myFunc(x):
  if x < 18:
    return False
  else:
    return True
a=list(filter(myFunc,ages))
print(a)                                    # [18, 24, 32]

adults = filter(myFunc, ages)
for x in adults:
  print(x)
```

output-

```
18
24
32
```

Reduce()

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.

This function is defined in “functools” module.

Working :

-At first step, first two elements of sequence are picked and the result is obtained.

-Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.

-This process continues till no more elements are left in the container.

-The final returned result is returned and printed on console.

```
import functools                    # importing functools for reduce()


lis = [1, 3, 5, 6, 2, ]              # initializing list

print ("The sum of the list elements is : ", end="")

print (functools.reduce (lambda a, b: a+b, lis))
```

output -   The sum of the list elements is : 17

```
# using reduce to compute maximum element from list

print("The maximum element of the list is : ", end="")

print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

output-   The maximum element of the list is : 6

```
# python code to demonstrate working of reduce()
# using operator functions

# importing functools for reduce()
import functools

# importing operator for operator functions
import operator

# initializing list
lis = [1, 3, 5, 6, 2, ]


# using reduce to compute sum of list
# using operator functions
print("The sum of the list elements is : ", end="")
print(functools.reduce(operator.add, lis))



# using reduce to compute product
# using operator functions
print("The product of list elements is : ", end="")
print(functools.reduce(operator.mul, lis))



# using reduce to concatenate string
print("The concatenated product is : ", end="")
print(functools.reduce(operator.add, ["geeks", "for", "geeks"]))
```

output –

```
The sum of the list elements is : 17

The product of list elements is : 180

The concatenated product is : geeksforgeeks
```

Decorator

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of a function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it. But before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

First Class Objects

In Python, functions are first class objects which means that functions in Python can be used or passed as arguments.

A function is an instance of the Object type.

You can store the function in a variable.

You can pass the function as a parameter to another function.

You can return the function from a function.

You can store them in data structures such as hash tables, lists, …

Consider the below examples for better understanding.

Example 1: Treating the functions as objects.

```
# Python program to illustrate functions

# can be treated as objects


def shout(text):

        return text.upper()

print(shout('Hello'))                    #HELLO

yell = shout

print(yell('Hello'))                    #HELLO
```

Example 2: Passing the function as an argument

```
# Python program to illustrate functions
# can be passed as arguments to other functions
def shout(text):
   return text.upper()

def whisper(text):
   return text.lower()

def greet(func):
   # storing the function in a variable
   greeting = func("""Hi, I am created by a function passed as an argument.""")
   print (greeting)

greet(shout)
greet(whisper)
```

Output:

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```

Example 3: Returning functions from another function.

```
# Python program to illustrate functions
# Functions can return another function

def create_adder(x):
   def adder(y):
      return x+y

   return adder

add_15 = create_adder(15)

print(add_15(10))
```

Output:

```
25
```

Decorators

As stated above the decorators are used to modify the behaviour of function or class. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax for Decorator:

```
@gfg_decorator
def hello_decorator():
    print("Gfg")


Above code is equivalent to -


def hello_decorator():
    print("Gfg")


hello_decorator = gfg_decorator(hello_decorator)
```

```python
# defining a decorator

def hello_decorator(func):


    def inner1():
        print("Hello, this is before function execution")


        func()

        print("This is after function execution")

    return inner1


def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behaviour
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```

Output:

```
Hello, this is before function execution

This is inside the function !!

This is after function execution
```

```
def function1():
    print("subscribe now")
func2=function1
func2()     #subscribe now
```

```
def fun(aa):
    if aa==0:
        return print
    if aa==11:
        return sum
a=fun(0)
print(a)        #<built-in function print>
a=fun(11)
print(a)       #<built-in function sum>
```

```
def executor(func):
    func("okokok")
executor(print)         #okokok
```

```
def dec1(func1):
    def nowexec():
        print("Executing now")
        func1()
        print("Executed")
    return nowexec

def who_is_harry():
    print("Harry is a good boy")
who_is_harry = dec1(who_is_harry)
who_is_harry()
```

output –

```
 Executing now
 Harry is a good boy
 Executed
```

```
def dec1(func1):
def dec1(func1):
    def nowexec():
        print("Executing now")
        func1()
        print("Executed")
    return nowexec
@dec1
def who_is_harry():
    print("Harry is a good boy")
who_is_harry()
```

output –

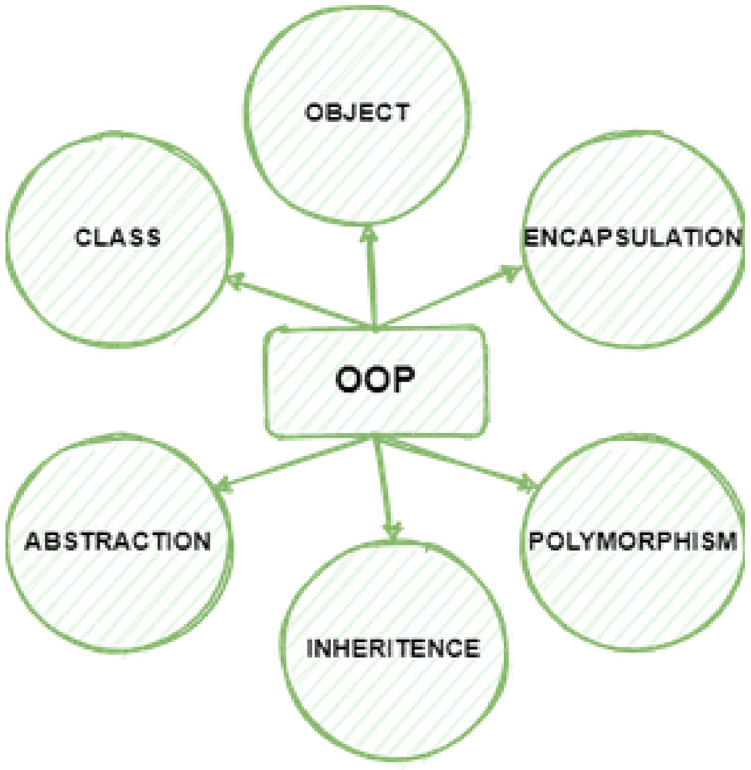
```
 Executing now
 Harry is a good boy
 Executed
```

OOPs concepts in python

The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

Main Concepts of Object-Oriented Programming (OOPs)

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



CLASS (OOPs)

A class is a collection of objects.

A class contains the blueprints or the prototype from which the objects are being created.

It is a logical entity that contains some attributes and methods.

Class = Template hota hai .

```
class  Dog:
   pass
```

we have created a class named “dog” using the “class” keyword.

OBJECT (OOPs)

The object is an entity that has a state and behavior associated with it.

Object = instance of class .

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating a object

```
obj = Dog()
```
create an object named “obj” of the class “Dog”

```
class student :                    #we made a class called STUDENT
    pass
harry=student()                     #we made object called harry
larry=student()                    #we made object called larry

harry.name="harry"           #instance variable of harry
harry.age=23                    #instance variable of harry
harry.std=12                     #instance variable of harry

larry.section="c"                                    #instance variable of larry
larry.sub=["hindi","english","maths"]          #instance variable of larry

print(harry.name , larry.sub ,harry.age,harry.name)
```

output -

```
harry ['hindi', 'english', 'maths'] 23 harry
```

Instance variable and class variable(OOPs)

```
 class Employee:                  #we create a class called Employee
   no_of_leaves=8             #we create a class variable
   pass
harry = Employee()             #we create object called harry
rohan =Employee()             #we create object called rohan
harry.name="harry"            # instance variable of harry
harry.salary=455                # instance variable of harry
harry.role="instructor"        # instance variable of harry
rohan.name="rohan"             #instance variable of rohan
rohan.salary=4566               #instance variable of rohan
rohan.role="student"           #instance variable of rohan
 print(Employee.no_of_leaves)       #8
 print(harry.no_of_leaves)              #8
rohan.no_of_leaves=9

print(rohan.__dict__)
output - {'name': 'rohan', 'salary': 4566, 'role': 'student', 'no_of_leaves': 9}

Employee.no_of_leaves=10
print(Employee.__dict__)
output - {'__module__': '__main__', 'no_of_leaves': 10, '__dict__': <attribute
'__dict__' of 'Employee'
 objects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>,
'__doc__': None}
```

Self Keyword (OOPs)

self represents the instance of the class.

```
class Employee:
    no_of_leaves=8

    def printdetails(self):                    #we create a method called printdetails
        return f"Name is {self.name} Salary is {self.salary} and role is
{self.role}"

harry = Employee()
rohan =Employee()

harry.name="harry"
harry.salary=455
harry.role="instructor"
rohan.name="rohan"
rohan.salary=4566
rohan.role="student"

print(rohan.printdetails())
```

output - Name is rohan Salary is 4566 and role is student

```
print(harry.printdetails())
```

output - Name is harry Salary is 455 and role is instructor

__init__ method constructor (OOPs)

```
class Employee:

    def __init__(self,a_name , a_salary , a_role):
        self.name= a_name
        self.salary= a_salary
        self.role= a_role

harry=Employee("HARRY","1500","SCIENTIST")

print(harry.salary)      #1500
print(harry.role)        #SCIENTIST
print(harry.name)        #HARRY
```

Class method (OOPs)

it can modify a class variable that will be applicable to all the instances.

```
class Employee:
    no_of_leaves = 8

    def __init__(self,a_name , a_salary , a_role):

        self.name= a_name
        self.salary= a_salary
        self.role= a_role


    @classmethod                       #change class variable
    def change_leaves(cls,newLeaves):
        cls.no_of_leaves=newLeaves


harry=Employee
harry.change_leaves(50)
print(harry.no_of_leaves)         #50
```

Class method as alternative constructors (OOPs)

```
class Employee:
    no_of_leaves = 8

    def __init__(self,a_name , a_salary , a_role):

        self.name= a_name
        self.salary= a_salary
        self.role= a_role

    @classmethod

    def from_str(cls,string):

        # return cls(*string.split("-"))              #u can write shortly to

        a=string.split("-")
        return cls(a[0],a[1],a[2])

karan=Employee.from_str("karan-1200-student")

print(karan.no_of_leaves)        #8
print(karan.salary)              #1200
```

Static method decorator (OOPs)

We generally use static methods to create utility functions.

```
class Employee:
    no_of_leaves = 8

    def __init__(self,a_name , a_salary , a_role):
        self.name= a_name
        self.salary= a_salary
        self.role= a_role

    @classmethod
    def from_str(cls,string):
        return cls(*string.split("-"))                #u can write shortly to

    @staticmethod
    def printgood(string):
        print("this is good "+ string)

karan=Employee.from_str("karan-1200-student")

karan.printgood ("harry")                #this is good harry
```

In the below example we use a static method to check if a person is an adult or not.

```
from datetime import date
class Person:
        def __init__(self, name, age):
                self.name = name
                self.age = age
       @classmethod
        def fromBirthYear(cls, name, year):
                return cls(name, date.today().year - year)

        # a static method to check if a Person is adult or not.
        @staticmethod
        def isAdult(age):
                return age > 18
person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print(person1.age)               #21
print(person2.age)               #25
print(Person.isAdult(22))        #True
```

The difference between the Class method and the static method (OOPs) :

- A class method takes cls as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

Data Abstraction (OOPs)

It hides the unnecessary code details from the user.

Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved through creating **abstract classes.**

Encapsulation (OOPs)

Encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from being accessed by other classes.

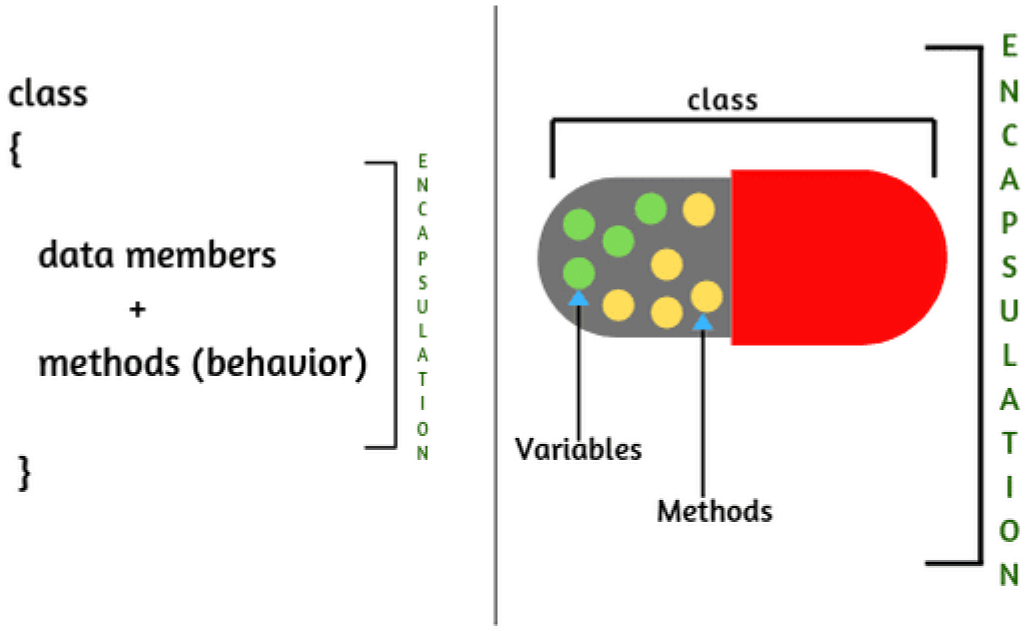


Fig: Encapsulation

Inheritance (OOPs)

Inheritance is the capability of one class to derive or inherit the properties from another class.

The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.

The benefits of inheritance are:

- It provides the reusability of a code. We don't have to write the same code again and again.
- Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance –

Single Inheritance

Multilevel Inheritance

Hierarchical Inheritance:

Multiple Inheritance:

Single Inheritance

When a class inherits another class, it is known as a single inheritance

Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

(Single inheritance- One class inherit with only one class)

SINGLE INHERITANCE

```
class Employee:
    no_of_leaves = 8

    def __init__(self,aname,asalary,arole):
        self.name=aname
        self.salary=asalary
        self.role=arole


    def prinntdetails(self):
        return f"the name is {self.name} . salary is {self.salary} . role is {self.role}"

    @classmethod
    def change_leaves(cls,newleaves):
        cls.no_of_leaves=newleaves
    @classmethod
    def fromdash(cls,string):
        return cls(*string.split("-"))



class Programmer(Employee):
    def __init__(self,aname,asalary,arole,alanguages):
        self.name = aname
        self.salary = asalary
        self.role = arole
        self.languages=alanguages


       def printpro(self):
        return f"the programmers name is {self.name} . salary is {self.salary} " \
               f". role is {self.role} . the languages are {self.languages}"



harry=Employee("HARRY",255,"INSTRUCTOR")
rOHAN=Employee("rohan",455,"student")
shubham=Programmer("SHUBHAM",555,"PROGRAMMER",["python"])
kARAN=Programmer("KARAN", 777,"PROGRAMMER",["python","c"])

print(kARAN.printpro())
print(kARAN.prinntdetails())
print(harry.prinntdetails())
```

output

```
the programmers name is KARAN . salary is 777 . role is PROGRAMMER . the languages are ['python',
'c']
the name is KARAN . salary is 777 . role is PROGRAMMER
the name is HARRY . salary is 255 . role is INSTRUCTOR
```

MULTIPLE INHERITANCE

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

```
class Employee:
    no_of_leaves = 8

    def __init__(self,aname,asalary,arole):
        self.name=aname
        self.salary=asalary
        self.role=arole


    def prinntdetails(self):
        return f"the name is {self.name} . salary is {self.salary} . role is {self.role}"

    @classmethod
    def change_leaves(cls,newleaves):
        cls.no_of_leaves=newleaves
    @classmethod
    def fromdash(cls,string):
        return cls(*string.split("-"))
    @staticmethod
    def printgood(string):
        print("this is good"+ string)

class Player:
    no_of_games=4

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

    def prinntdetails(self):
        return f"the name is {self.name} . game is {self.game}"

class Coolprogrammer(Employee,Player):
    language="c++"

    def printlanguage(self):
        print(self.language)

shubham=Player("SHUABHAM",["CRICKET"])

karan=Coolprogrammer("KARAN",55555,"COOLPROG")
a=karan.prinntdetails()
karan.printlanguage()
print(a)
```

output

```
c++
the name is KARAN . salary is 55555 . role is COOLPROG
```

Multilevel Inheritance

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

```
class Dad:
    basketball=1


class Son(Dad):

    dance=1
    def isdance(self):
        return f"Yes i dance {self.dance} no of times"


class Grandson(Son):

    dance=6
    def isdance(self):
        return f"Yes i dance awesomely {self.dance} no of times"


darry = Dad()
larry= Son()
harry=Grandson()

print(harry.isdance())  #Yes i dance awesomely 6 no of times
print(harry.basketball)   #1
```

Access Modifiers in Python : Public, Private and Protected (OOPs)

- Python control access modifications which are used to restrict access to the variables and methods of the class
- Python uses '_' (underscore) symbol to determine the access control for a specific data member or a member function of a class
- Access specifiers in Python have an important role to play in securing data from unauthorized access.

A Class in Python has three types of access modifiers:

- **Public Access Modifier**
- **Protected Access Modifier**
- **Private Access Modifier**

Public Access Modifier

The members of a class that are declared public are easily accessible from any part of the program.

```
class Geek:

     # constructor
     def __init__(self, name, age):

           # public data members
           self.geekName = name
           self.geekAge = age

     # public member function
     def displayAge(self):

           # accessing public data member
           print("Age: ", self.geekAge)

# creating object of the class
obj = Geek("R2J", 20)

# accessing public data member
print("Name: ", obj.geekName)

# calling public member function of the class
obj.displayAge()
```

Output:

```
Name:  R2J

Age:  20
```

In the above program, geekName and geekAge are public data members and displayAge() method is a public member function of the class Geek. These data members of the class Geek can be accessed from anywhere in the program.

Protected Access Modifier

The members of a class that are declared protected are only accessible to a class derived from it.

Data members of a class are declared protected by adding a single underscore ‘_’ symbol before the data member of that class.

```
# super class
class Student:

   # protected data members
   _name = None
   _roll = None
   _branch = None

   # constructor
   def __init__(self, name, roll, branch):
      self._name = name
      self._roll = roll
      self._branch = branch

   # protected member function
   def _displayRollAndBranch(self):

      # accessing protected data members
      print("Roll: ", self._roll)
      print("Branch: ", self._branch)
```

```
# derived class
class Geek(Student):

     # constructor
     def __init__(self, name, roll, branch):
          Student.__init__(self, name, roll, branch)

     # public member function
     def displayDetails(self):

  # accessing protected data members of super class
          print("Name: ", self._name)

 # accessing protected member function of super class
          self._displayRollAndBranch()

# creating objects of the derived class
obj = Geek("R2J", 1706256, "Information Technology")

# calling public member functions of the class
obj.displayDetails()
```

Output:

```
Name:  R2J

Roll:  1706256

Branch:  Information Technology
```

In the above program, _name, _roll, and _branch are protected data members and _displayRollAndBranch() method is a protected method of the super class Student. The displayDetails() method is a public member function of the class Geek which is derived from the Student class, the displayDetails() method in Geek class accesses the protected data members of the Student class.

Private Access Modifier

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

```
class Geek:

    # private members
    __name = None
    __roll = None
    __branch = None

    # constructor
    def __init__(self, name, roll, branch):
        self.__name = name
        self.__roll = roll
        self.__branch = branch

    # private member function
    def __displayDetails(self):

        # accessing private data members
        print("Name: ", self.__name)
        print("Roll: ", self.__roll)
        print("Branch: ", self.__branch)
```

```
    # public member function
    def accessPrivateFunction(self):

        # accessing private member function
        self.__displayDetails()

# creating object
obj = Geek("R2J", 1706256, "Information
Technology")

# calling public member function of the class
obj.accessPrivateFunction()
```

```
output :

Name:  R2J

Roll:  1706256

Branch:  Information Technology
```

In the above program, __name, __roll and __branch are private members, __displayDetails() method is a private member function (these can only be accessed within the class) and accessPrivateFunction() method is a public member function of the class Geek which can be accessed from anywhere within the program. The accessPrivateFunction() method accesses the private members of the class Geek.

Below is a program to illustrate the use of all the above three access modifiers (public, protected, and private) of a class in Python:

```
# super class
class Super:

    # public data member
    var1 = None

    # protected data member
    _var2 = None

    # private data member
    __var3 = None

    # constructor
    def __init__(self, var1, var2, var3):
        self.var1 = var1
        self._var2 = var2
        self.__var3 = var3

   # public member function
    def displayPublicMembers(self):

        # accessing public data members
        print("Public Data Member: ", self.var1)

    # protected member function
    def _displayProtectedMembers(self):

        # accessing protected data members
        print("Protected Data Member: ", self._var2)

    # private member function
    def __displayPrivateMembers(self):

        # accessing private data members
        print("Private Data Member: ", self.__var3)

    # public member function
    def accessPrivateMembers(self):

        # accessing private member function
        self.__displayPrivateMembers()
```

```
# derived class
class Sub(Super):

    # constructor
     def __init__(self, var1, var2, var3):
          Super.__init__(self, var1, var2, var3)

    # public member function
     def accessProtectedMembers(self):

          # accessing protected member functions of super class
          self._displayProtectedMembers()

# creating objects of the derived class
obj = Sub("Geeks", 4, "Geeks !")

# calling public member functions of the class
obj.displayPublicMembers()
obj.accessProtectedMembers()
obj.accessPrivateMembers()

# Object can access protected member
print("Object is accessing protected member:", obj._var2)

# object can not access private member, so it will generate Attribute error
#print(obj.__var3)
```

Output -

```
Public Data Member:  Geeks

Protected Data Member:  4

Private Data Member:  Geeks !
```

In the above program, the accessProtectedMembers() method is a public member function of the class *Sub* accesses the _displayProtectedMembers() method which is protected member function of the class Super and the accessPrivateMembers() method is a public member function of the class Super which accesses the __displayPrivateMembers() method which is a private member function of the class Super.

Polymorphism (OOPs)

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

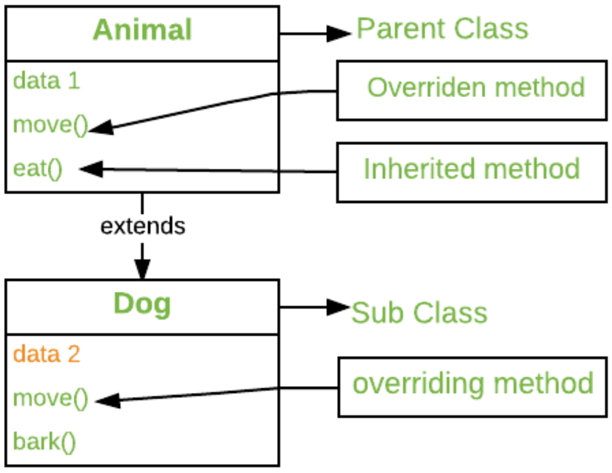
Example -

```
# len() being used for a string
print(len("geeks"))                    #5

# len() being used for a list
print(len([10, 20, 30]))               #3
```

Overriding and Super () (OOPs)

Overriding

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.



```
# method overriding
```

```
# Defining parent class
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)


# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

Output:

```
Inside Parent

Inside Child
```

Super()

When we want to call an already overridden method, then the use of the super function comes in.

```
   class A:
   classvar1="i ama a class variable in class a"               #class variable
   def __init__(self):
      self.var1="i am inside class A constructor"

class B(A):                                                #CLASS B INHERIATE IN CLASS A
   classvar2="i am in class B"                              #class variable
a=A()
b=B()
print(b.classvar1)   # output - i ama a class variable in class a
```

```
class A:
   classvar1 = "i ama a class variable in class a"                  #class variabel

   def __init__(self):
      self.var1 = "i am inside class A constructor"
      self.classvar1="instance var in class A"                  #instance variable

class B(A):
   classvar2 = "i am in class B"                              #class variable
a = A()  #instance
b = B()  #instance
print(b.classvar1)              # output - instance var in class A
```

```
class A:
    classvar1 = "i ama a class variable in class a"                    #class variabel

    def __init__(self):
        self.var1 = "i am inside class A constructor"
        self.classvar1="instance var in class A"                       #instance variable
        self.special="special"
class B(A):
    classvar1 = "i am in class B"                                #class variable

    def __init__(self):
        self.var1 = "i am inside class B constructor"
        self.classvar1 = "instance var in class B"
a = A()  #instance
b = B()  #instance
print(b.special)   #shoe error coz override ko read ni krta , uska koi bajud ni hota , special ko
print ni krega
```

```
class A:
    classvar1 = "i ama a class variable in class a"                    #class variabel

    def __init__(self):
        self.var1 = "i am inside class A constructor"
        self.classvar1="instance var in class A"                       #instance variable
        self.special="special"
class B(A):
    classvar1 = "i am in class B"                                #class variable

    def __init__(self):
        super().__init__()
        self.var1 = "i am inside class B constructor"
        self.classvar1 = "instance var in class B"
a = A()  #instance
b = B()  #instance
print(b.special)    #output - special  (we use super().__init__() to print super class  )
```

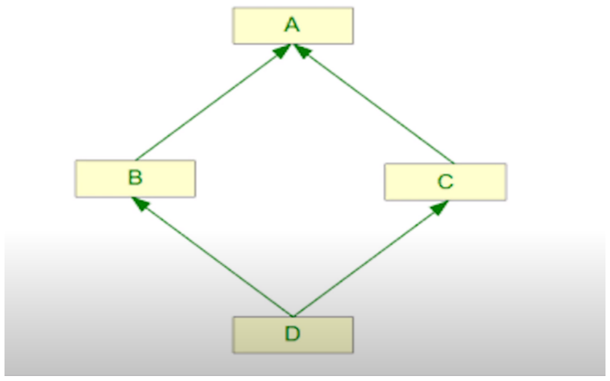
```
class A:
    classvar1 = "i ama a class variable in class a"                    #class variabel

    def __init__(self):
        self.var1 = "i am inside class A constructor"
        self.classvar1="instance var in class A"                    #instance variable
        self.special="special"
class B(A):
    classvar1 = "i am in class B"                              #class variable

    def __init__(self):
        super().__init__()
        self.var1 = "i am inside class B constructor"
        self.classvar1 = "instance var in class B"
        super().__init__()
a = A()  #instance
b = B()  #instance
print(b.special,b.var1,b.classvar1)
#special i am inside class A constructor instance var in class A
```

```
class A:
    classvar1 = "i ama a class variable in class a"                    #class variabel

    def __init__(self):
        self.var1 = "i am inside class A constructor"
        self.classvar1="instance var in class A"                    #instance variable
        self.special="special"
class B(A):
    classvar1 = "i am in class B"                              #class variable

    def __init__(self):
        super().__init__()
        self.var1 = "i am inside class B constructor"
        self.classvar1 = "instance var in class B"
        super().__init__()
        print(super().classvar1)
a = A()  #instance
b = B()  #instance
print(b.special,b.var1,b.classvar1)
output -
i ama a class variable in class a
special i am inside class A constructor instance var in class A
```

Diamond shape problem in Multiple inheritance (OOPs)

Diamond shape problem

It is about priority related confusion, which arises when four classes are related to each other by an inheritance relationship, as shown in the image below:



In the above image, we can see that class C and class B are inheriting from class A, or it can be said as class A is a parent to class B and C. And class D is inheriting from both class C and B. So, in a way, they are all in relation to one and other somehow. Let us write down the relation in code format so it will be easier to understand.

```
class A:

pass

class B(A):

pass

class C(A):

pass

class D( B, C ):

pass
```

1. If we have a method that is only present in class A and we use the class D object to call the method, it will go to class A while checking for the method name in all the classes in between and run the method in class A.
2. However, if the same method is also present in class B, then it will run the B class method because, for class D, class B holds more priority than class A. The reason is that class D is derived from class B, which is further derived from A. So, a closer relation exists with B than A.
3. If the same method is present in classes C and B, it may create a little bit of confusion. But as we have already discussed in Tutorial #61, that in such cases, our priority is based from left to right, meaning whichever class is on the left side will be given more priority, and then we will move towards the right one. In this case, the left class is B, so the method in B will de be executed first.

```
class A:
   def met(self):
      print("this is a method from class A")
class B(A):
   pass
class C(A):
   pass
class D(B,C):
   pass
a=A()
b=B()
c=C()
d=D()

d.met()    #op - this is a method from class A

class A:
   def met(self):
      print("this is a method from class A")
class B(A):
   def met(self):
      print("this is a method from class B")
class C(A):
   def met(self):
      print("this is a method from class C")
class D(B,C):
   def met(self):
      print("this is a method from class D")
a=A()
b=B()
c=C()
d=D()

d.met() #this is a method from class D
c.met() #this is a method from class C
b.met() #this is a method from class B
```

Operator Overloading In Python (OOPs)

Operator overloading is the process of using an operator in different ways depending on the operands. You can change the way an operator in Python works on different data-types.

A very popular and convenient example is the **Addition (+) operator**. Just think how the '+' operator operates on two numbers and the same operator operates on two strings. It performs **"Addition"** on numbers whereas it performs **"Concatenation"** on strings.

Dunder or magic methods in Python (OOPs)

Dunder here means "Double Under (Underscores)" . Dunder or magic methods in Python are the methods having two prefix and two suffix underscores in the method name.

These are commonly used for operator overloading.

Few examples for magic methods are: `__init__` , `__add__` , `__len__` , `__repr__` etc.

__repr__ = representation of your class objects using the __repr__ method , __repr__ is a special method used to represent a class's objects as a string.

__str__ = The __str__ method in Python **represents the class objects as a string** – it can be used for classes. The __str__ method should be defined in a way that is easy to read and outputs all the members of the class.

__init__ =The init method is used to create an instance of the class.

```
  class employee:

    def __init__(self,aname,asalary,arole):
        self.name=aname
        self.salary=asalary
        self.role=arole

    def __add__(self, other):         #dunder add methor is used for operator overloading
        return self.salary + other.salary
emp1=employee("harry",355,"programmer")
emp2=employee("rohan",5,"cleaner")
print(emp1+emp2)        #360
```

```
class employee:

    def __init__(self,aname,asalary,arole):
        self.name=aname
        self.salary=asalary
        self.role=arole

    def __truediv__(self, other):        #dunder truediv methor is used for operator overloading
        return self.salary / other.salary
emp1=employee("harry",355,"programmer")
emp2=employee("rohan",5,"cleaner")
print(emp1/emp2)        #71.0
```

```
class employee:

    def __init__(self,aname,asalary,arole):
        self.name=aname
        self.salary=asalary
        self.role=arole

    def __repr__(self):
        return f"employee('{self.name} , {self.salary},{self.role}')"
emp1=employee("harry",355,"programmer")
print(emp1)          #employee('harry , 355,programmer')
```

```
class employee:

    def __init__(self,aname,asalary,arole):
        self.name=aname
        self.salary=asalary
        self.role=arole

    def __str__(self):
        return f"the name is {self.name} . salary is {self.salary} and role is {self.role}"
emp1=employee("harry",355,"programmer")
print(emp1)            #the name is harry . salary is 355 and role is programmer
```

__call__ (OOPs)

The `__call__` method enables Python programmers to write classes where the instances behave like functions and can be called like a function. When the instance is called as a function

Example 1:

```
class Example:
    def __init__(self):
        print("Instance Created")


    # Defining __call__ method
    def __call__(self):
        print("Instance is called via special method")
  # Instance created
e = Example()
# __call__ method will be called
e()
```

Output :

```
Instance Created
Instance is called via special method
```

Example 2:

```
class Product:
    def __init__(self):
        print("Instance Created")

    # Defining __call__ method
    def __call__(self, a, b):
        print(a * b)

# Instance created
ans = Product()

# __call__ method will be called
ans(10, 20)
```

Output :

```
Instance Created
200
```

Abstract Base Class & @abstractmethod (OOPs)

An abstract class is a class that holds an abstract method.

An abstract method is a method defined inside an abstract class.

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class.

Why use Abstract Base Classes :

By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses.

To implement an abstract class, we have to import the abc module by using an import statement. Along with it, we have to import the abstract method too

from abc import ABC, abstractmethod

syntex

```
from abc import ABC, abstractmethod
Class MyClass(ABC):
    @abstractmethod
    def mymethod(self):
        #empty body
        pass
```

```
# from abc import ABCMeta, abstractmethod
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def printarea(self):
        return 0

class Rectangle(Shape):
    type = "Rectangle"
    sides = 4
    def __init__(self):
        self.length = 6
        self.breadth = 7

    def printarea(self):
        return self.length * self.breadth

rect1 = Rectangle()
print(rect1.printarea())

output
42
```

```
from abc import ABC, abstractmethod
 class Polygon(ABC):
  @abstractmethod
    def noofsides(self):
        pass

 class Triangle(Polygon):
    def noofsides(self):
        print("I have 3 sides")

 class Pentagon(Polygon):
    def noofsides(self):
        print("I have 5 sides")

 class Hexagon(Polygon):
    def noofsides(self):
        print("I have 6 sides")

 class Quadrilateral(Polygon):
    def noofsides(self):
        print("I have 4 sides")



 R = Triangle()
 R.noofsides()

 K = Quadrilateral()
 K.noofsides()

 R = Pentagon()
 R.noofsides()

 K = Hexagon()
 K.noofsides()
```

Output:

```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

Property Decorator (getter, setter, deleted, and Doc)

Decorators are functions that take another function as an argument, and their purpose is to modify the other function without changing it.

- Property decorator is a built-in function in Python.
- Property decorator is a pythonic way to use getters and setters in object-oriented programming, which comes from the Python property class.
- Property decorator is composed of four things, i.e., getter, setter, deleted, and Doc. The first three are methods, and the fourth one is a docstring or comment.

Use @property along with the getter method to access the value of the attribute.

`fget` is a function for retrieving an attribute value.

`fset` is a function for setting an attribute value.

`fdel` is a function for deleting an attribute value.

`doc` creates a docstring for attribute

```
class Employee:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
        # self.email = f"{fname}.{lname}@codewithharry.com"

    def explain(self):
        return f"This employee is {self.fname} {self.lname}"

    @property
    def email(self):
        if self.fname==None or self.lname == None:
            return "Email is not set. Please set it using setter"
        return f"{self.fname}.{self.lname}@codewithharry.com"

    @email.setter
    def email(self, string):
        print("Setting now...")
        names = string.split("@")[0]
        self.fname = names.split(".")[0]
        self.lname = names.split(".")[1]

    @email.deleter
    def email(self):
        self.fname = None
        self.lname = None
hindustani_supporter = Employee("Hindustani", "Supporter")
print(hindustani_supporter.email)  #Hindustani.Supporter@codewithharry.com

hindustani_supporter.fname = "US"
print(hindustani_supporter.email)  #US.Supporter@codewithharry.com

hindustani_supporter.email = "this.that@codewithharry.com"
print(hindustani_supporter.fname) op - setting not , this

del hindustani_supporter.email
print(hindustani_supporter.email)

output - Email is not set. Please set it using setter

hindustani_supporter.email = "Harry.Perry@codewithharry.com"
print(hindustani_supporter.email)
# output - Setting now... , Harry.Perry@codewithharry.com
```

Code introspection in Python(OOPs)

Introspection is an ability to determine the type of an object at runtime

Everything in python is an object.

Every object in Python may have attributes and methods.

By using introspection, we can dynamically examine python objects.

Code Introspection is used for examining the classes, methods, objects, modules, keywords and get information about them so that we can utilize it.

Introspection reveals useful information about your program's objects.

Python provides some built-in functions that are used for code introspection.	
type()	This function returns the type of an object
dir()	function return list of methods and attributes associated with that object.
str()	This function converts everything into a string .
id()	This function returns a special id of an object.
help()	It is used it to find what other functions do
hasattr()	Checks if an object has an attribute
getattr()	Returns the contents of an attribute if there are some.
repr()	Return string representation of object
callable()	Checks if an object is a callable object (a function)or not.
issubclass()	Checks if a specific class is a derived class of another class.
isinstance()	Checks if an objects is an instance of a specific class.
sys()	Give access to system specific variables and functions
__doc__	Return some documentation about an object
__name__	Return the name of the object.

type() : This function returns the type of an object.

```
import math

print(type(math))          <class 'module'>


print(type(1))               <class 'int'>



print(type("1"))             <class 'str'>



rk =[1, 2, 3, 4, 5, "radha"]

print(type(rk))               <class 'list'>


print(type(rk[1]))            <class 'int'>


print(type(rk[5]))            <class 'str'>
```

.dir() : This function return list of methods and attributes associated with that object.

```
import math
rk =[1, 2, 3, 4, 5]


print(dir(rk))
```

```
output - ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

```
rk =(1, 2, 3, 4, 5)
print(dir(rk))
```

```
output - ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count',
'index']
```

```
rk ={1, 2, 3, 4, 5}
print(dir(rk))
print(dir(math))
```

```
output - ['__and__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__iand__', '__init__', '__init_subclass__',
'__ior__', '__isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
'__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference',
'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop',
'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

```
output - ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter',
'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

str() : This function converts everything into a string .

```
a = 1
print(type(a))              <class 'int'>


# converting integer into string
a = str(a)
print(type(a))                <class 'str'>


s =[1, 2, 3, 4, 5]
print(type(s))              <class 'list'>


# converting list into string
s = str(s)
print(type(s))                <class 'str'>
```

id() : This function returns a special id of an object.

```
import math
a =[1, 2, 3, 4, 5]
# print id of a
print(id(a))                   139787756828232

b =(1, 2, 3, 4, 5)
# print id of b
print(id(b))                   139787756828232

139787757942656

c ={1, 2, 3, 4, 5}
# print id of c
print(id(c))                   139787757391432
```

Iterable , iterator , iteration , GENERATORS

ITERABLE –

An **iterable** is an object that has an __iter__ method which returns an **iterator**, or which defines a __getitem__ method that can take sequential indexes starting from zero .
So an **iterable** is an object that you can get an **iterator** from.

ITERATOR –

An **iterator** is an object with a next (Python 2) or __next__ (Python 3) method.

ITERATION –

Iteration is a general term for taking each item of something, one after another. Any time you use a loop, explicit or implicit, to go over a group of items, that is iteration.

```
>>> s = 'cat'     # s is an ITERABLE
                  # s is a str object that is immutable
                  # s has no state
                  # s has a __getitem__() method

>>> t = iter(s)    # t is an ITERATOR
                   # t has state (it starts by pointing at the "c"
                   # t has a next() method and an __iter__() method

>>> print (next(t))      # the next() function returns the next value and advances the state
'c'
>>>print( next(t))       # the next() function returns the next value and advances
'a'
>>> print(next(t))       # the next() function returns the next value and advances
't'
>>>print ( next(t))      # next() raises StopIteration to signal that iteration is complete
Traceback (most recent call last):

StopIteration

>>> iter(t) is t                # the iterator is self-iterable
```

Generators -

Generators concept is also very similar as it is used to make an iterator.

The only difference comes in the return statement.

The generator does not use a return statement. Instead, it uses a **yield** keyword.

Yield functionality is very similar to return as it returns a value to the caller, but the difference is that it also saves the state of the iterator.

Meaning that when we use the function again, the yield will resume the value from the place it left off.

Generators in Python are created just like the normal functions using the 'def' keyword.

Generator functions do not run by their name, and they are run when the __next__() function is called.

A generator is very helpful in projects relating to memory issues because, like a simple iterator, it does not return all the values at a time; instead, it produces, series of values over time. So a generator is generally used when we want to iterate over a series of values but do not want to store them completely in memory.

```
def getNum (x):
   for i in range(x):
      yield i


seq = getNum (2)
print(seq.__next__())
print(seq.__next__())
print(seq.__next__())
```

output –

```
0
1
Traceback (most recent call last):
File "<stdin>", line 7, in <module>
 print(seq.__next__())
StopIteration
```

When we run print(seq.__next__()) for the third time, StopIteration is raised. This is because a for loop takes an iterator and iterates over it using __next__() function, which automatically ends when StopIteration is raised.

Applications : Suppose we to create a stream of Fibonacci numbers, adopting the generator approach makes it trivial; we just have to call next(x) to get the next Fibonacci number without bothering about where or when the stream of numbers ends.
A more practical type of stream processing is handling large data files such as log files. Generators provide a space efficient method for such data processing as only parts of the file are handled at one given point in time. We can also use Iterators for these purposes, but Generator provides a quick way (We don’t need to write __next__ and __iter__ methods here.

Comprehensions in Python

Comprehensions in Python can be defined as the Pythonic way of writing code. Using comprehension, we compress the code so it takes less space. Comprehension in Python converts the four to five lines of code into a one-liner.

Comprehension's importance comes in the scenarios when the project is too big, for example, Google is made up of 2 billion lines of code, Facebook is made from 62 million lines of code, Windows 10 has roughly 50 million lines of code. So in such scenarios, comprehension has to be implemented as much as we can so that the lines of code decreases and the efficiency increases.

list

```
a=[]
for i in range(50):
   if i %3==0:
      a.append(i)
print(a)                    #[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
a =[i for i in range(50) if  i % 3==0]
print(a)                 # [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

dictionary

```
d={i:f"item{i}" for i in range (5)}
print(d)                          #{0: 'item0', 1: 'item1', 2: 'item2', 3: 'item3', 4: 'item4'}
```

generator

```
evens = (i for i in range(100) if i%2==0)
print(evens.__next__())                #0
print(evens.__next__())                #2
```

Using Else With For Loops

We have to write an else statement using the Else keyword, followed by a colon after the for loop block of code.

```
Syntax:

for x in n:

  #statements

 else:

  #statements
```

When we use **else** with **for** loop, the compiler will only go into the **else** block of code when two conditions are satisfied:

The loop is normally executed without any error.

We are trying to find an item that is not present inside the list or data structure on which we are implementing our loop.

Except for these conditions, the program will ignore the else part of the program. For example, if we are just partially executing the loop using a break statement, then the else part will not execute. So, a break statement is a must if we do not want our compiler to execute the else part of the code.

For Example:

```
for i in ['C','O','D','E']:

 print(i)

else:

print("Statement successfully executed")
```

Output:

```
C

O

D

E

Statement successfully executed
```

```
khana = ["roti", "Sabzi", "chawal"]
for item in khana:
     print(item)
else:
    print("Your item was not found")
    output
    roti
    Sabzi
    chawal
    Your item was not found
```

```
khana = ["roti", "Sabzi", "chawal"]
for item in khana:
    if item =="partha"
        break
else:
    print("Your item was not found")        #your item was not found.
```

Function caching

Caching means storing the data in a place from where it can be served faster.

In the case of data that has been frequently used, the computer assigns a cache memory, so it does not have to load it again and again from the main memory.

The purpose of the cache is to make the tasks more efficient and quicker.

The same is true for web browsers; the pages we load again and again are stored in the cache for faster retrieval.

In Python, however, we have to do it all manually, as the program will not store anything in the cache itself.

How to use function caching in Python?

Function caching is a way to improve code's performance by storing the function's return values. Before the 3.2 updates of Python, we had to create a cache ourselves by storing the value in a variable or by other such means. But in Python 3.2, there is a new update in the functools module of Python. To use this module, we have to import it first

Functools modules

The **functools** module in Python deals with higher-order functions, that is, functions operating on(taking as arguments) or returning functions and other such callable objects.

The functools module provides a wide array of methods such as cached_property(func), cmp_to_key(func), lru_cache(func), wraps(func), etc.

lru_cache()

lru_cache() is one such function in **functools** module which helps in reducing the execution time of the function by using memoization technique.

Without funtool

```
import time
def somework (n):
   time.sleep(n)
   return  n

if __name__ ==
'__main__':
   print("ok")
   somework(3)
   print("okk")
   somework(3)
   print("ddd")
```

With funtool module

```
import time
from functools import lru_cache
@lru_cache(maxsize=32)
def somework (n):
   time.sleep(n)
   return  n

if __name__ == '__main__':
   print("ok")
   somework(3)
   print("okk")
   somework(3)
   print("ddd")
```

output

```
ok
okk
ddd
```

increase the time and save cach of program , so he program run faster

```
import time
from functools import lru_cache


@lru_cache(maxsize=32)
def some_work(n):
    #Some task taking n seconds
    time.sleep(n)
    return n


if __name__ == '__main__':
    print("Now running some work")
    some_work(3)
    some_work(1)
    some_work(6)
    some_work(2)
    print("Done... Calling again")
    input()
    some_work(3)
    print("Called again")
```

Try, Except, Else and Finally in Python

An Exception is an Event, which occurs during the execution of the program.

It is also known as a **run time error**.

When that error occurs, Python generates an exception during the execution and that can be handled, which avoids your program to interrupt.

Exception handling with try, except, else, and finally

- Try: This block will test the excepted error to occur
- Except: Here you can handle the error
- Else: If there is no exception then this block will be executed
- Finally: Finally block always gets executed either exception is generated or not

- In the try block, all the statements are executed until an exception occurs.
- Except block is used to catch and handle the exception(s) that occurs during the execution of the try block.
- Else block runs only when no exceptions occur in the execution of the try block.

Finally block always runs; either an exception occurs or not.

understood by the table below:

Try	Not running	Running
Except	Will run	Will not run
Else	Will not run	Will run
Finally	Will run	Will run

```
a = 5
b = 0
print(a/b)
```

Output:

```
Traceback (most recent call last):
  File "/home/8a10be6ca075391a8b174e0987a3e7f5.py", line 3, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

In this code, The system can not divide the number with zero so an exception is raised

```
# Python code to illustrate working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        result = x // y
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")

# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```

Output:

```
Yeah ! Your answer is : 1
Sorry ! You are dividing by zero
```

Else Clause

The code enters the else block only if the try clause does not raise an exception.

```
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)


divide(3, 2)
divide(3, 0)
```

Output:

```
Yeah ! Your answer is : 1
Sorry ! You are dividing by zero
```

Finally Keyword

Python provides a keyword finally, which is always executed after try and except blocks.

```
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        print('This is always executed')


divide(3, 2)
divide(3, 0)
```

Output:

```
Yeah ! Your answer is : 1
This is always executed


Sorry ! You are dividing by zero
This is always executed
```

Coroutines in python

Coroutines are mostly used in cases of time-consuming programs, such as tasks related to machine learning or deep learning algorithms,

or in cases where the program has to read a file containing a large number of data.

In such situations, we do not want the program to read the file or data again and again, so we use coroutines to make the program more efficient and faster.

Coroutines run endlessly in a program because they use a while loop with a true or 1 condition, so it may run until infinite time. Even after yielding the value to the caller, it still awaits further instruction or calls.

We have to stop the execution of the coroutine by calling the coroutine.close() function. It is crucial to close a coroutine because its continuous running can take up memory space,

When you call a coroutine, nothing happens. They only run in response to the next() and send() methods.

Coroutine requires the use of the next statement first so it may start its execution. Without a next(), it will not start executing. We can search a coroutine by sending it the keywords as input using object name along with send(). The keywords to be searched are send inside the parenthesis.

When we run the **next() function** the first time, the coroutine executes and waits for new input. After the input is sent to it using the send() function, it executes it and again waits for next input, and the process goes on like this because we have set the while loop as true, so it will never exit its execution. In order to make the execution stop, we have to close the coroutine using coroutine.close() function.

- **send() — used to send data to coroutine**
- **close() — to close the coroutine**
- **Next () – to start the execution**

```
def myfunc():
    print("Code With Harry")
    while True:
        value = (yield)
        print(value)

coroutine =myfunc()
next(coroutine)
coroutine.send("Python")
coroutine.send(" Tutorial ")
coroutine.close()
```

output -

```
Code With Harry
 Python
 Tutorial
```

```
def searcher():
   import time
   # Some 4 seconds time consuming task
   book = "This is a book on harry and code with harry and good"
   time.sleep(4)
     while True:
      text = (yield)
      if text in book:
         print("Your text is in the book")
      else:
         print("Text is not in the book")
```

```
a = searcher()
print("search started")
next(a)
print("Next method run")
a.send("harry")
a.close()
a.send("harry")
```

Output –

```
search started
Next method run
Your text is in the book
#after close() code is not execuated
```

```
def bare_bones():

    print("My first Coroutine!")

    try:
        while True:
            value = (yield)
            print(value)
    except GeneratorExit:
        print("Exiting coroutine...")

coroutine = bare_bones()

next(coroutine)

coroutine.send("First Value")

coroutine.send("Second Value")

coroutine.close()
```

output –

```
My first Coroutine!
First Value
Second Value
Exiting coroutine...
```

```
def filter_line(num):
    while True:
        line = (yield)
        if num in line:
            print(line)

cor = filter_line("33")
next(cor)
cor.send("Jessica, age:24")
cor.send("Marco, age:33")
cor.send("Filipe, age:55")


output –

"Marco, age:33"
```

OS Module

OS module provides our code with a direct connection to the operating system.

We can use its different functions to interact and do activities on our operating system.

For example, if we want to create such software that needs to store or access files from a directory or folder, we can use the OS module to perform the task for us.

To use OS Module in Python, we have to import it.

OS modules have a lot of built-in functions.

Built-in Functions	**Working**
print(dir(os))	It gives us a list of all the functions the OS module is composed of.
os.getcwd( ):	Here cwd is a short form for the current working directory. The function returns us the path of the directory we are currently in. It is important to know about our directory because when we are trying to import a file in python, the compiler searches for it in our current directory.
os.chdir( ):	It is used in case we want to change our directory. The new path is sent inside the parenthesis. If we want to access some files or folders from some other directory, we can use it.
os.listdir( ):	If we want to output the names of all the directories at a certain location, we can use this function.
os.mkdir( ):	To create a new directory or folder. The name is sent as a parameter inside the parenthesis.
os.makedirs( ):	To make more than on directory simultaneously.

os.rename( ):	To rename an already existing directory, we use this. We send the current name and new name of our directory as parameters.
os.rmdir( ):	It deletes an empty directory.
os.removedirs( ):	We can remove all directories within a directory by using removedirs().
os.environ.get( ):	It will return us the environment variable. The environment variable must be set, or the function will return null.
os.path.join( ):	It joins one or more path components. We can join the paths by simply using a + sign, but the benefit of using this function is that we do not have to worry about extra slashes between the components. So less accuracy still provides us with the same result.
os.path.exists( ):	It returns us a Boolean value, i.e., either true or false. It is used to check whether a path exists or not. If it does, then the output will be true, otherwise false.
os.path.isfile( ):	It returns true if the path is an existing regular file.
os.path.isdir( ):	It returns true if the path is an existing directory.

```
import os
# print(dir(os))
# print(os.getcwd())
# os.chdir("C://")
# print(os.getcwd())
# f = open("harry.txt")
# print(os.listdir("C://"))
# os.makedirs("This/that")
# os.rename("harry.txt", "codewithharry.txt")
# print(os.environ.get('Path'))
# print(os.path.join("C:/", "/harry.txt"))


# print(os.path.exists("C://Program Files2"))
print(os.path.isfile("C://Program Files"))
```

The main purpose of the OS module is to interact with the operating system.

The primary use of the OS module is to create folders, remove folders, move folders, and sometimes change the working directory

Request Modules In Python

The requests module allows you to send HTTP requests using Python.

The requests module is not a built-in Python module; instead, we have to download it manually. Requests module is used to send all kinds of HTTP requests.

The HTTP request returns a Response Object with all the response data (content, encoding, status, etc.

What is HTTP ?

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML.

It was designed for communication between web browsers and web servers, but it can also be used for other purposes

It is a set of protocols that are designed to enable communication between clients and servers. Between clients and servers, it works as a request-response protocol. To request a response from the server, we can request data from the server or submit data to be processed to the server.

Syntax

```
requests.methodname(params)
```

Method	Description
delete(url, args)	Sends a DELETE request to the specified url
get(url, params, args)	Sends a GET request to the specified url
head(url, args)	Sends a HEAD request to the specified url
patch(url, data, args)	Sends a PATCH request to the specified url
post(url, data, json, args)	Sends a POST request to the specified url
put(url, data, args)	Sends a PUT request to the specified url
request(method, url, args)	Sends a request of the specified method to the specified url

```
import requests

x = requests.get('https://w3schools.com/python/demopage.htm')

print(x.text)


OUTPUT –
<!DOCTYPE html>
<html>
<body>
<h1>This is a Test Page</h1>
</body>
</html>
```

JSON Modules (JavaScript Object Notation)

Python has a built-in package called json .

JSON stands for JavaScript Object Notation.

Java Script Object Notation (JSON) is a light weight data format with many similarities to python dictionaries.

JSON objects are useful because browsers can quickly parse them, which is ideal for transporting data between a client and a server.

If we convert a JSON string to Python, OR PYTHON OBJECT TO JSON , the conversion is also known as parsing, and that is the keyword we use professionally for the conversion.

What parsing actually does?

Parsing converts the code from one form to another, making it compatible with running on the other platform by changing all the little syntax differences and making it perfect for running on the other platform. The following table shows how Python objects are converted to JSON objects.

JSON	PYTHON
true	true
false	false
string	string
number	number
array	list, tuple
object	dict
null	none

If you have a JSON string, you can parse it by using the json.loads() method.

If you have a Python object, you can convert it into a JSON string by using the json.dumps() method.

Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the json.loads() method.

The result will be a Python dictionary.

```
import json
# some JSON:
x =  '{ "name":"John", "age":30, "city":"New York"}'
# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])

output
30
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the json.dumps() method.

```
import json
# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)
# the result is a JSON string:
print(y)
```

output

```
{"name": "John", "age": 30, "city": "New York"}
```

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

Output -

```
{"name": "John", "age": 30}
["apple", "bananas"]
["apple", "bananas"]
"hello"
42
31.76
true
false
null
```

```
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

Output –

```
{"name": "John", "age": 30,
"married": true, "divorced":
false, "children":
["Ann","Billy"], "pets": null,
"cars": [{"model": "BMW
230", "mpg": 27.5},
{"model": "Ford Edge",
"mpg": 24.1}]}
```

Pickle Module (built in module)

The pickle module is used for implementing binary protocols for serializing and de-serializing a Python object structure.

What is pickling?

Pickling is the process of serializing an object. Serializing means storing the object in the form of binary representation so it can be saved in our main memory. The object could be of any type. It could be a string, tuple, or any other sort of object that Python supports. The data is stored in the main memory in a file. While writing the code for pickling, we open the file in "wb" mode, also known as writing binary mode. So, to use the pickle module, we have to make a file with the .pkl extension and send it in a dump() function along with the object. dump() is a built-in function in the Pickle module, made for pickling.

To use pickle, we have to import it in Python.

import pickle

What is unpickling?

The file format is binary, so we cannot directly open and read it; instead, we have to open it using a python program, and the process is known as unpickling. We have to open the file in "rb" mode for unpickling, also known as a read binary mode. The function we use this time is also a built-in function, named load(). Unlike dump(), we have to send the file name as a parameter, and it automatically retrieves the data in the object type it was inserted in. For example, if we send a list while pickling, the return result will also be a list.

We can face some of the common pickle exceptions raised while dealing with the pickle module.

- **Pickle.PicklingError:** If the pickle object does not support pickling, then Pickle.PicklingError exception is raised.

- **Pickle.UnpicklingError:** This exception will raise if the file contains bad or corrupted data.

- **EOF Error:** This exception will be raised if the end of the file is detected.

Disadvantages:

- There are some situations in which pickling is discouraged. For example, when we are working with multiple programming languages, pickle is discouraged.
- Pickle has been found slower than its alternatives.
- In some cases, it has also shown a few security vulnerabilities.
- When we update our program to a newer version, pickled data through the previous version can cause issues.

```
import pickle

# syntax
pickle.dump(object , file object )

                                   #pickling
cars= ["audi","bmw"]

#we made a file mycar.pkl
file="mycar.pkl"

#now we made a file object
fileobject=open(file,'wb')          # ‘wb’ writing binary mode

pickle.dump(cars,fileobject)

fileobject.close()
```

output -
mycar.pkl file will made in wb form , open mycar.pkl in text form ,file is not readable .

#de-pickling

```
file = "mycar.pkl"
fileobject = open(file,'rb')
mycar = pickle.load(fileobject)
print(mycar)
print(type(mycar))
```

output -

```
['audi', 'bmw']
<class 'list'>
```

Regular Expressions Modules (RegEx)

A Regular Expressions (RegEx**)** is a special sequence of characters that uses a search pattern to find a string or set of strings.

It can detect the presence or absence of a text by matching it with a particular pattern, and also can split a pattern into one or more sub-patterns.

Python provides a **re** module that supports the use of regex in Python.

Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

```
import re
```

The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"
()	Capture and group	

special sequence

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	**Description**	**EXAMPLE**
\A	Returns a match if the specified characters are at the beginning of the string -	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) is present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character

The findall() Function

The findall() function returns a list containing all matches.

```
import re
#Return a list containing every occurrence of "ai":

txt = "The rain in Spain"

x = re.findall("ai", txt)

print(x)
```

OUTPUT - ['ai', 'ai']

Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"

#Check if "Portugal" is in the string:

x = re.findall("Portugal", txt)

print(x)

if (x):

  print("Yes, there is at least one match!")

else:

  print("No match")
```

OUTPUT - [ ]

```
                No match
```

```
# A Python program to demonstrate working of
# findall()
import re

# A sample text string where regular expression
# is searched.
string = """Hello my Number is 123456789 and
            my friend's number is 987654321"""

# A sample regular expression to find digits.
regex = '\d+'

match = re.findall(regex, string)
print(match)
```

Output

```
['123456789', '987654321']
```

search() Function

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

```
import re

txt = "The rain in Spain"

x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())

output –

The first white-space character is located in position: 3
```

Make a search that returns no match:

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)

output –

none
```

split() Function

The split() function returns a list where the string has been split at each match:

Split at each white-space character:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

output –

```
['The', 'rain', 'in', 'Spain']
```

You can control the number of occurrences by specifying the maxsplit parameter:

Example

Split the string only at the first occurrence:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

output –

```
['The', 'rain', 'in', 'Spain']
```

sub() Function

The sub() function replaces the matches with the text of your choice:

Replace every white-space character with the number 9:

```
import re
#Replace all white-space characters with the digit "9":
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x) output -The9rain9in9Spain
```

```
import re
#Replace the first two occurrences of a white-space character with the digit
9:
txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
output –
The9rain9in Spain
```

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value None will be returned, instead of the Match Object.

Do a search that will return a Match Object:

```
import re

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) #this will print an object
```

output -█<_sre.SRE_Match object; span=(5, 7), match='ai'>

The Match object has properties and methods used to retrieve information about the search, and the result:

.span() - returns a tuple containing the start-, and end positions of the match.
.string - returns the string passed into the function
.group() - returns the part of the string where there was a match

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

```
import re

#Search for an upper case "S" character in the beginning of a word, and print its position:

txt = "The rain in Spain"

x = re.search(r"\bS\w+", txt)

print(x.span())
```

output - (12, 17)

Print the string passed into the function:

```
import re


#The string property returns the search string:


txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
output -The rain in Spain
```

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re


#Search for an upper case "S" character in the beginning of a word, and
print the word:


txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

output – Spain

Converting .py file to .exe file

In windows, .exe is an extension used for executable files. It is one of the most common file extensions used for almost all kinds of software and applications programs and setups.

Now let's come to the purpose of the conversion.

We create lots of Python programs and want to share them with the world. If we want our python program to run on any computer without the IDE or even installing Python itself, we must convert it to .exe.

All the apps and programs we use on our computer are written using some language or code, but we do not have to install the particular language for the program to run.

Steps to Create an Executable from Python Script using Pyinstaller

Step 1: Install the Pyinstaller Package

The first step is to install the module pyinstaller. For this, we will create or destinate a folder and open our power shell window there using shift + mouse right-click.

Now we will run the following command for our module installation.

```
pip install pyinstaller
```

Make a txt file called main . write a program in it

```
print("which are the number you want  to add")
print(int(input()) + int(input()))
```

clt + s to save

open window powershell click shift + right mouse button

write – python .\main.txt

next type to convert txt file to exe filr

pyinstaller .\main.txt

two folder are made in folder dict and build

open dict - main – click main to open main.exe file

click shift +right mouse open window powershell

Once we click on the file, we are ready to launch our program. So, that's how you can easily convert the .py file to the .exe file in Python.

Raise (keyword)

Python raise Keyword is used to raise exceptions or errors

To throw (or raise) an exception, use the `raise` keyword.

```
x = 5

if x < 7:
  raise Exception("Sorry, no numbers below zero")
```

output - Traceback (most recent call last):

```
  File "C:\Users\amart\PycharmProjects\pythonProject\pythonProject4\raise keyword.py", line 4, in <module>

    raise Exception("Sorry, no numbers below zero")

Exception: Sorry, no numbers below zero
```

```
x = 5

if x < 4:
  raise Exception("Sorry, no numbers below zero")
```

outout - ……………………………………………….

Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

output – Traceback (most recent call last):

```
  File "./prog.py", line 4, in <module>

TypeError: Only integers are allowed
```

Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(7) is int:
  raise TypeError("Only integers are allowed")
```

output –

…………………………………

```
a= input("what is your name ")4
if a.isnumeric():
  raise Exception("number is not allowed")

output -
what is your name 4
Traceback (most recent call last):
  File "C:\Users\amart\PycharmProjects\pythonProject\pythonProject4\raise keyword.py", line 8, in
<module>
    raise Exception("number is not allowed")
Exception: number is not allowed
```

```
a= input("what is your name ")
b = input("how much to you earn ")
if int(b)==0:
  raise ZeroDivisionError("b is 0 stopping the program ")
print(f"hello {a}")

output – gaveoutput
harry
4
Hello harry
```

```
c = input("Enter your name")
try:
    print(a)

except Exception as e:

    if c =="harry":
        raise ValueError("Harry is blocked he is not allowed")

    print("Exception handled")
output gave –
Enter your nameharry
Traceback (most recent call last):
  File "C:\Users\amart\PycharmProjects\pythonProject\pythonProject4\raise keyword.py", line 26, in <module>
    print(a)
NameError: name 'a' is not defined

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "C:\Users\amart\PycharmProjects\pythonProject\pythonProject4\raise keyword.py", line 31, in <module>
    raise ValueError("Harry is blocked he is not allowed")
ValueError: Harry is blocked he is not allowed
```

== vs is difference between == , is

'==' is used to represent value equality.
Value equality means two variables or objects or even data structures such as a list composed of the same value.
Suppose we have two variables, a and b.
We assign the value 2 to both of them.
Now, as we know that they do not have any direct link with each other.
The only similarity is that they have been given the same value.
So, if we place an '==' sign between them, the output will be True.
However, when we change the value of one of the variables, it will return false.

For Example:-

```
x = [1, 2, 3, 4]

y= [1, 2, 3, 4]

x == y

output = True


x = [1, 2, 3, 4]

y= [0, 8, 7, 7]

x == y

output = False
```

An equality operator (==)expression evaluates to True if the objects referred to by the variables have the same contents.

The identity operator 'is' tracks the object back to its identity while the equality operator '==' only compares the values.

Is = An identity operator(is) expression evaluates to True if two variables point to the same object.

```
a= [7,8,9]

b=a

print(b is a)
```

output

```
True                (it is true coz they are point same object )
```

```
# == - value equality - Two objects have the same value

# is - reference equality - Two references refer to the same object



# Task:

a =[6, 4 , "34"]

b = [6, 4 , "34"]

print(b is a)
```

Creating a Command Line Utility In Python

What is command line

The command line is **a text interface for your computer**. It's a program that takes in commands, which it passes on to the computer's operating system to run. From the command line, you can navigate through files and folders on your computer

What is a Command Line Interface(CLI)?

A command-line utility is a way of giving operating system instructions using lines of text. Command-line programs operate via the command line or PowerShell. It will interact with a command-line script.

A command-line interface or command language interpreter (CLI), also known as command-line user interface, console user interface, and character user interface (CUI), is a means of interacting with a computer program where the user (or client) issues commands to the program in the form of successive lines of text (command lines)

Advantages of CLI:

- Requires fewer resources
- Concise and powerful
- Expert-friendly
- Easier to automate via scripting

Why to use CLI in your python program?

Now let us come to why we should use the command-line utility in our program.

We can easily call a command line program in Python into a different language program.

Each program has calling support in it for calling the command lines program.

So in cases, where we are writing a program in some other language, but we want to perform a task in Python and call it in our program, then the command line can help us do that.

अब हम इस बात पर आते हैं कि हमें अपने प्रोग्राम में कमांड-लाइन यूटिलिटी का उपयोग क्यों करना चाहिए।

हम पाइथन में एक कमांड लाइन प्रोग्राम को आसानी से एक अलग भाषा प्रोग्राम में कॉल कर सकते हैं।

कमांड लाइन प्रोग्राम को कॉल करने के लिए प्रत्येक प्रोग्राम में कॉलिंग सपोर्ट होता है।

तो ऐसे मामलों में, जहां हम किसी अन्य भाषा में प्रोग्राम लिख रहे हैं, लेकिन हम पायथन में एक कार्य करना चाहते हैं और इसे अपने प्रोग्राम में कॉल करना चाहते हैं, तो कमांड लाइन हमें ऐसा करने में मदद कर सकती है।

For creating a Command Line Utility In Python

first import two modules, i.e., argsparse and sys.

Argsparse - helps us to get command-line arguments in our program

sys - module helps us to import the code we wrote using argparse onto the console.

Import argsparse

Import sys

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language

```
import argparse
import sys

def calc(args):
    if args.o == 'add':
        return args.x + args.y

    elif args.o == 'mul':
        return args.x * args.y

    elif args.o == 'sub':
        return args.x - args.y

    elif args.o == 'div':
        return args.x / args.y

    else:
        return "Something went wrong"

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--x', type=float, default=1.0,
                        help="Enter first number. This is a utility for calculation.
Please contact harry bhai")

    parser.add_argument('--y', type=float, default=3.0,
                        help="Enter second number. This is a utility for calculation.
Please contact harry bhai")

    parser.add_argument('--o', type=str, default="add",
                        help="This is a utility for calculation. Please contact harry
bhai for more")

    args = parser.parse_args()
    sys.stdout.write(str(calc(args)))
```

Contents