Chapter 1: Programming Language

* you can select any script or oop's based languages.

But most popular languages for backend are

1. JavaScript 2. Java 3. Python.

* As python syntax and readability is easy we lear python Programing language.

Phython:

→ 1) Installation: o install python from python.org and setup environment for python. Check if it is installed in command prompt by "python --version"

exercise 1

```
# printing statements.

print ("Hello World")
print ("Hello Again")
print ("yay! printing")
print ('I "said" do not touch this.')
```

How do you get your country's language characters into my file.

Ans: Make sure you type this at the top of your file.

of your file : `# -*- coding: utf-8 -*-`

Numbers and Maths

`+ - / * % < > <= >=`

ex2.py

```
print ("I will now count my chickens:")
print ("Hens", 25 + 30/6)
print ("Roosters", 100 - 25 * 3 % 4)

print ("Now I will count the eggs:")
print (3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6)
print ("Is it true that 3 + 2 < 5 - 7 ?")
print (3 + 2 < 5 - 7)
print ("What is 3 + 2 ?", 3 + 2)
```

Output:

```
I will now count my chickens:
Hens 30
Roosters 97
Now I will count the eggs:
7
Is it true that 3+2<5-7?
False.
What is 3+2? 5.
```

Questions:

→ What is order of operations?

In the US we use an acronym called PEMDAS, which stands for parantheses Exponents Multiplication Division Addition Subtraction. That's the order python follows as well.

→ Why does / divide round down?

It's not really rounding down, it's just dropping the fractional part after the decimal. Try doing 7.0 / 4.0 and compare it to 7/4 and you'll see the difference.

Note: for rounding a floating value we can use round() function like this: round(1.7333).

Taking input dynamically:

```
age = int (input (" Enter your age :" ))
height = int ( input (" Enter your height :" ))
print (" you are %v old and %v tall ) % (age, height) )
```

(or)

```
print (" you are ", age ," old and ", height, " tall ").
```

Parameters, Unpacking, Variables

you can pass variables to a script (script being another name of your .py file).

you will see command line arguments input as

example:

```
from sys import argv
script , first, second , third = argv
print (" The script is called :", script)
print (" your first variable is :", first)
print (" your second variable is :", second)
print (" your third variable is :", third).
```

In terminal : python ex9.py 10 20 30

Reading Files

This exercise involves writing two files. One is your usual ex11.py file that you will run, but the other is named ex11_sample.txt. This second file isn't a script but a plain text file we'll be reading in our script.

Here are the contents of that file:

```
This is stuff I typed into a file.
It is really cool stuff
Lots and lots of fun to have in here.
```

" We will discuss later".

Taking input in python

By 1. input (prompt)

2. raw_input (prompt).

input - This function taken the input from the user, and converts it into a string. The type of returned object always will be <class 'str'>

example:

```
val= input (" Enter your value:")
Print (val).
```

There are various functions that are desired input for of them are:-

- int(input()) → float(input())

ex:-

```
num = int(input("Enter a number:"))
print(num," ",type(num))
float Num = float(input("Enter a decimal number:"))
print(floatNum," ",type(floatNum))
```

Taking multiple inputs at a time

using split() method:-

ex 12.py

```
a,b = input("Enter two values:").split()
print("First num is {} and second num is {}".format(a,b))

# taking multiple line inputs at a time
# and type casting using list() function.
x = list(map(int, input("Enter multiple values:").split()))
print("list of students :",x)
```

using List comprehension:

ex13.py

```
# how to take multiple inputs
# using List comprehension.

# taking two inputs at a time
x, y = [int(x) for x in input("Enter two values: ").split()]
print(" First name is {} and second name is {}".format(x,y))

# taking multiple line inputs at a time.
x = [int(x) for x in input(" Enter multiple values: ").split(",")]
print(" Number of list is :", x)
```

Python print() function

String Literals:-

\n : The string literal is used to add a new blank line while printing a statement.

"" : An empty quotes ("") is used to print empty line

end=" " : This is used to print in same line we can print like end="**" so that blw two print() statements ** will be printed.

print Concatenated strings :- print ("Hemant" + "sai" + "nag")

Output formatting :- a, b = 10, 1000

Print ("The value of a is {} and b is {}".format(a, b))

sep = " " . a = 12
b = 12
c = 2024
Print (a, b, c, sep = "-") output : 12-12-2024

It should be always at last of print statement

print (a, b, c, sep = "-") ✓

print (a, b, sep = "-", c) ✗

printing without newline in python 3.x without using for loop

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

print (*l)

print without using newline using Python sys module.

import sys
sys.stdout.write ("Geeks are powerful")
sys.stdout.write (" is best website for coding!")

using format() method and referring a position of the object.

```
print ('{0} and {1}'.format ("Geeks', 'Portal'))

# combine positional and keyword arguments.
print ("Number one portal is {0}., {1} and {other}.'
    .format ('Geeks', 'for', other = 'Geeks')).
```

Python Operators

Typer of operators in Python

1. Arthmetic Operators
2. Comparison Operators.
3. Logical operators.
4. Bitwise Operators.
5. Assignment Operators.
6. Identity Operators and Membership Operators.

Arthmetic Operators

Operators	Description	Syntax
+	addition	x+y
-	subtraction	x-y
*	Multiplication	x*y
/	division (float)	x/y
//	Divison (floor):	x//y
%o	giver remainder	x%oy
**	Power	x**y

PEMDAS Paranthesis Exponentiation Multiplication. division Addition subtraction.

Comparision operators

>, <, ==, !=, >=, <=

= is an assignment operator and == comparison operator.

Logical Operators in python

Operator	description	Syntax
and	Logical AND : True if both the operands are true.	x and y
or	Logical or x y output 0 0 0 1 0 1 0 1 1 1 1 1	x or y
not	True if operand is false.	not x.

Bitwise operators

&	and	x & y
|	OR	x | y
~	NOT	~x
^	XOR	x ^ y
>>	right shift	x >>
<<	left shift	x <<

* assignment operator " = " x = y + z

* we can write x = x + y as x += y

a = a - b as a -= b

a = a % b as a %= b etc ...

Identity Operators in Python

is - True if the operands are identical

is not - True if the operands are not identical.

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

output : True
True.

Membership Operators

in - True if value is found in the sequence.

not in - True if value is not found in the sequence.

ex :

```
x = 24
y = 20
list = [10, 20, 30, 40, 50]
if (x not in list):
    print (" x is not present in given list").
```

```
if (y in list)
    print ("y is present in given list")
```

Ternary Operator

```
a, b = 10, 20
min = a if a<b else b
print (min).
```

any()

any returns true if any of the items is True and returns False if empty or all are false:

#example

```
print ( any ( [False, False, False, False]) // false
print ( any ( [False, True, False, False]) // True
print ( any ( [True, False, False, False]) // True.
```

Copy

all()

all is reverse of any.

{ paste

output :)

```
print (all ([True, True, True, True ]))    // True
print (all ([False, True, False, False]))  // False
print (all ([True, False, False, False]))  // False
```

for example :-

```
list1 = [ ]
list2 = [ ]
for i in range (1,21)
    list1.append (4*i-3)
for i in range (0,20)
    list2.append (list1[i] %2 == 1)
print ("see whether all numbers in list1 are odd =>")
print (all (list2))
```

// Output : True

Operator Functions in Python

```
import operator

a = 4
b = 3

print ( operator. add (a,b))

print ( operator. sub(a,b))

print ( operator. mul(a,b))

print ( operator. truediv(a,b))   // division with float

print ( operator. floordiv(a,b))

print ( operator. pow(a,b))

print ( operator. mod (a,b))

if ( operator . lt (a,b)) :   // less than  a<b

if ( operator. le (a,b)) :   // less than or equal  a<=b

if ( operator. eq (a,b)) :  // equals ==  (a==b)

if (  "     . gt(a,b)) :  // greater than  a>b

if (  "     . ge(a,b)) :  //  "  equal  a>=b

if (  "     . ne (a,b)) :  // not equal  a!=b.
```

String methods (exercise - ex 16.py)

*
```
name = "Bro Code"
print(len(name))  // 10
print(name.find("o"))  // 2
print(name.capitalize())  // Bro code
print(name.upper())  // BRO CODE
print(name.lower())  // bro code
print(name.isdigit())  // False
print(name.isalpha())  // False: because space between bro code
print(name.count("o"))  // 2   if "Brocode" then true.
print(name.replace("o", "a"))  // Bra Code
print(name * 3)  // Bro code Bro code Bro code
```

Math functions (ex15-py)

```
import math

pi = 3.14

print(round(pi))  // 3
print(math.ceil(pi))  // 4
print(math.floor(pi))  // 3
print(abs(pi))  // 3.14  if -3.14 it changes to 3.14.
print(pow(pi,2))  //
print(math.sqrt(420))  // 20.4939.
```

#

```
x = 1
y = 2
z = 3
print (max(x, y, z))    // 3
print ( min(x, y, z))   // 1
```

String slicing :- ex14.py

slicing = create a substring by extracting elements from another string indexing[] or slice()

[start : stop : step].

```
name = "Hemanth Sai Nag"
       0 1 2 3 4 5 6.
first_name = name[0 : 6+1 = 7]
print( first_name):
last_name = name[8 : 15] or name[8 :]
print (last_name).
funky_name = name [0 : 15 : 2]   (or)   name [ : : 3]
reversed_name = name[ : : -1]
print ( reversed_name)   # gaN iaS htnameH.

website1 = "https://google.com"
           0 1 2 3 4 5 6 7 8 9 10
           -4 -3 -2 -1
slice = slice(7, -4)
website2 = "https://wikipedia.com"
                    -4 -3 -2 -1
slice2 = slice(7, -4)
print (website1 [slice])
print (website2 [slice]).
```

If

if statement = a block of code that will execute if it's condition true.

```
age = int(input("How old are you?: "))

if age >= 18 :
    Print("you are an adult")
else:
    print("you are a child")
```

(or)

```
if {age >= 18}:
    print("you are an adult")
elif age < 0:
    print("you haven't been born yet!")
else:
    print("you are a child!")
```

While loop = (ex18.py)

```
name = None

while not name :
    name = input ("Enter your name:")
print ("Hello" + name)
```

for loops:- a statement that will execute it's block of code a limited amount of times.

(ex19.py)

while loop = unlimited

for loop = limited.

```
for i in range(10):
    print (i+1)

for i in range(50,100+1):          # prints 50 to 100.
    print (i)

for i in range(50,100+1,2):        # prints 2 step like 50,52,54,--
    print(i)

for i in "Bro Code":
    print(i).
```

```
import time
```

(ex20.py)

```
for seconds in range(10,0,-1):
    print(i)
    time.sleep(1)
print("Happy new year!").
```

Nested loops:- The "inner loop" will finish all of it's iterations before finishing one iteration of the "outer loop"

(ex21.py)

```
rows = int(input("How many rows"))
Columns = int(input("How many columns:"))
symbol = input("Enter a symbol to use:"))

for i in range(rows):
    for j in range(columns):
        print(symbol, end="").
```

break, continue, pass (ex22.py)

```
# break = used to terminate the loop entirely
# continue = skips to the next iteration of the loop.
# pass = does nothing, acts as a placeholder.
```

```
while True:
    name = input(" Enter your name: ")
    if name != "":
        break

phone_number = "123-456-7890"

for i in phone_number:
    if i == "-":
        continue
    print(i, end="")
e

for i in range(1,21)
    if i==13:
        pass
    else:
        print(i)
```

ex23.py = Program to find age in days.

ex24.py = Program to find roots of quadratic equation.

ex25.py = Program to find seconds in min, hours.

ex26.py = Program for swapping of 2 numbers. with 3rd variable.

ex27.py = Program for swapping of 2 numbers without 3rd variable.

ex28.py : swapping of 2 numbers using bitwise operators.

ex29.py : Program to find roots of quadratic equation using control flow.

ex30.py = Write a program to read person age and pincode in kakinada urban area pincode is 53303 and rural area pincode is 53304. Find the person is elligible for vote or note. If elligible find plan for voting.

ex31.py = Program to accept student details like student id, name, maths, physics, chemistry, find total, average, grade.

ex32.py = Program to print n natural numbers.

ex33.py = Program to print even numbers up to n

ex34.py = Program to print n natural numbers using time.

ex35.py = Program to convert foriengn heat into celsium from 0 to 300 and difference is 20.

ex36.py = Program to print multiplication table of given number unit 12 and until choice of continue is no.

ex37.py : Program to find sum of natural numbers upto n using while. loop.

ex38.py : Write a program to find factorial of n natural numbers.

ex39.py : Find power of xy without using math.pow() function.

ex40.py : Find sum of digits of given number

input : 245

Output : 2+4+5 = 11

ex41.py : Program to find reverse number of given number

ex42.py : Program to find given number is palindrom or not.

ex43.py : Program to find number is armstrong or not.

ex44.py : Program to find given number is prime or not

ex45.py : Program to find and print prime numbers upto n.

ex46.py : Program to find GCD of two numbers.

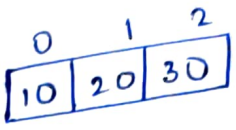
ex47.py : Program to print n terms of Fibonacci series.

Data structures in python

1.)

List:

ex1: l = [10, 20, 30]



ex2: a = [10, 2.5, 'Satya', True]

* Index is applicable.

In list the values are inserted based on index.

By default index starts from 0.

* It is mutable: we can change data which is in list
* slice is applicable: we can get particular group of values from list.
* Order is preserved: If we can insert values in list they are inserted in sequential order.
* It is growable.

List allows to insert values then it is increase the memory. List allows to delete value then it is decreases the memory.

* It allows duplicates.
* It is a heterogeneous.

It allows different datatypes at a time.

→ list can be nested, ex: ["sree", 2.0, 5, [30,40]]

Member ship operator in list:-

ex:
```
li = [10,20,4.5,"satyam's"]
check = 10 in li
print (check)    # True
check = 5.6 in li
print (check)    # False
check = 'TCS' not in li
print ( check)   # True.
```

Deleting items in list

ex:
```
cricketplayer = ["sachin", "dhoni", "virat"]
del cricketplayer [0]
print ( cricketplayer)
```

Output : ["dhoni", "virat"]

Methods used for List

append () : a method append adds single element to the end of list

insert() : a method insert to insert element at particular inden value.

extend() : a method extends to adds (join) in the list.

indun() : to find value in list if found returns inden value otherwise say value not found in list.

ex:

ex 48.py

```
num = [10, 20, 70]
print (num)
num. append ("hemanth")
num. append ("kkd")
print (num)
num. insert ( 3 , "sai")
Nonveg = [" chicken" , "eggs"]
num. extend (Nonveg)
print (num)
```

Searching in list

ex:-

```
l = [10, 20, 30]
l. index (10)      return 0
l. index (30)      return 2
l. index (50)      value not found in list.
```

Tuple :

1. It index is applicable
2. It is immutable (not changeable)
3. Slice. is applicable.
4. Order is preserved.
5. The value should be enclosed with in ()
6. It is not grow able.
7. It allows duplicates.
8. It is heterogeneous type.

ex:- company = ("infosys", "satyam", "IBM", "TCS")

vagrant = ("gold", True, 24, 2.0)

Slicing in list

```
                 0      1      2       3       4       5       6
months = ("Jan", "Feb", "Mar", "April", "May", "June", "July"
           7
"Aug", "sep", "Oct", "Nov", "Dec")

First quarter = months [0:3]

second quarter = months [3:6]
```

Output:- ("Jan", "Feb", "Mar")

("April", "May", "June")

Note:

If we can slice the tuple then it generates the list.

1. We can't update items in tuple.

   Tuples have no append or extend methods.

2. We can't remove items from a tuple.

   Tuple have no remove or pop methods.

3. We can't find items in tuple.

   Tuple has no index methods.

4. We can use in (membership) to see if an item is exist the tuple.

Note: Tuple faster than list. If you are defining a constan set of values better to choose tuple instead of list.

Dictionary:

1. A dictionary is an unordered unsorted set of key-value pairs.
2. When you add a key to a dictionary you must add a value for that key.
3. The key is always unique. if we use same key then overwriter the existing key.
4. The duplicate value is allowed.
5). It is represented as { }

Dictionary

Keys	Values
11	Infosys
22	Satyam
33	IBM
44	TCS

Creating dictionary

```
food = {'veg':'tamoto', 'nonveg':'chicken', 'sweet':'laddu'}
print (food)
type (food)
```

Output: {'veg':'tamoto', 'nonveg':'chicken', 'sweet':'laddu'}

Dictionary:

1. A dictionary is an unord. unsorted set of key-value pairs.
2. When you add a key to a dictionary you must add a value for that key.
3. The key is always unique. if we use same key then overwriter the existing key.
4. The duplicate value is allowed.
5). It is represented as {}

Dictionary

Keys	Values
11	Infosys
22	Satyam
33	IBM
44	TCS

Creating dictionary

```
food = {'veg':'tamoto', 'nonveg':'chicken', 'sweet':'laddu'}
print(food)
type(food)
```

Output: {'veg':'tamoto', 'nonveg':'chicken', 'sweet':'laddu'}

<class 'dict'>

Dictionary Operations:

```
inventory = {'apples': 20, 'bananas': 12, 'oranges': 24}
print (inventory) # {'apple': 20, 'bananas': 12, 'oranges'}

del inventory ['apples']
print (inventory) # {'bananas': 12, 'oranges': 24}

Inventory ['oranges': 50]
print (inventory) # {'bananas': 12, 'oranges': 50}
inventory ['mangoes'] = 25
print (inventory) # {'bananas': 12, 'oranges': 50, 'mangoes': 25}
len (inventory) # 3.
```

dictionary methods

len() : to find length of dictionary

keys() : to display keys in dictionary

values() : to display values in dictionary.

items() : to display both in the form of list of tuples, one for each.

copy() : creates duplicate dictionary.

Aliasing and Copying :-

```
opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
x = opposites
print(opposites['right'])  # wrong
y = opposites.copy()
x['right'] = 'left'
print(opposite['right'])  # left.
y['right'] = 'hemu'
print(opposites['right'])  # left.
```

opposite , x

up	down
right	left
true	false.

y

up	down
right	'hemu
true	false.

Sets :

1. It is same as list but
2. It dosent allows duplicates.
3. It is mutable
4. Order is not preserved (no order)
5. It is represented as { }
6. Index, slicing is not allowed bcz. it does not follow the order.

ex :- s = {10, 30, 50}.

Methods in sets

add() : to add item to a set

update() : to add set to a set

len() : to find length of set.

remove() : to remove item from set. If value not found gives Key Error.

discard() : to remove item from set. If found elu not not found ; it does nothing.

pop() : to remove item from begining in sequential order.

clear() : to clear all items from set, it returns empty set.

ex49.py

Ex:

```
s = {10,40,20,60,30}
print(s)
s.add(25)
print(s)
s.remove(40)
print(s)
s1 = {11,22}
s.update(s1)
print(s)
s.discard(22)
print(s)
s.pop()
s.clear()
print(s)
```

Output :

{20,40,10,60,30}

{20,40,25,60,30,10}

{20,25,10,60,30}

{20,22,25,10,11,60,30}

{20,25,10,11,60,30}

{25,10,11,60,30}

set()

Frozenset

exactly same as set but it is immutable.

```
s = {10, 20, 30, 40}
type(s)     # set
fs = frozenset(s)
type(fs)    # <class 'frozenset'>
fs.add(50).  # attribute error.
fs.remove(20)  # attribute error.
```

None:

→ The None type denotes a null object.

→ python provides exactly one null object, which is written as None in a program.

→ None means nothing i.e no associated value.

→ Its like as null in java.

```
>>> def f2():
        print("hello")

>>>f2()
hello.
>>> print(f2())
None
```

Python RegEx

Its primary function is to offer a search, where it takes a regular expression and a string. Here it either returns the first match or else none.

ex: import re

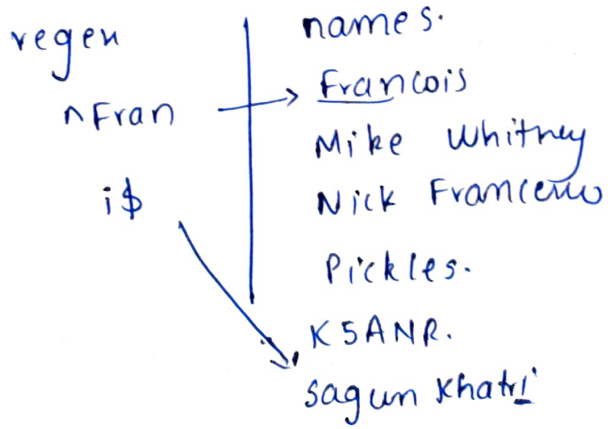
match = re.search(r'portal', 'Geeeks for Geeks: A computer science \ portal for geeks').

print (match)
print (match.group())
print ('start index:', match.start())
print (' End Index:', match.end())

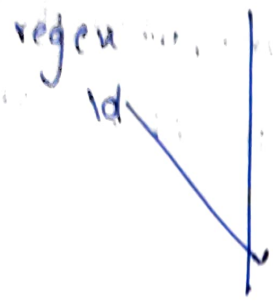
* ^ $ describes the position
* [a-t] , \w describes set of characters.
* + * ?q? -> Quantifiers.

^s.ta
s[a-t]*$
\w.?q?.

Regex position patterns:



Regex Character Sets.



\d : matches [0-9]

\s : matches white space

\w : matches a-z, A-Z, 0-9,

Phython Functions

function declaration:

keyword. function name. Parameter.

```
def function_name (parameters):
    # statement.  ---> Body of statement.
    return expression
```

Function return.

ex:

```
def fun():
    print("Hello")

fun()
```

ex:

```
def add(num1: int, num2: int) -> int:
    num3 = num1 + num2
    return num3

num1, num2 = 5, 10
rr = add(num1, num2)
rint(f"The addition of {num1} and {num2} results {ans}.")
```

ex:- ex50.py

```
def evenodd (x):
    if (x%2 == 0):
        print ("even")
    else:
        print ("odd")

evenodd (2)
evenodd (3).
```

Types Of Python function Arguments

1) default argument
2) Keyword arguments (named arguments)
3) Positional arguments.
4) Arbitrary arguments (variable - length arguments *args and **kwargs).

Default Argument ex51.py

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

ex:-

```
def myFun ( x, y=50):
    print("x:", x).
    print("y:", y)

myFun (10).
```

Keyword arguments

The idea is to allow the caller to specify the argument name with value so that the caller does not need to remember the order of parameters.

ex 52.py

```
def student ( firstname, lastname)
    print(firstname, lastname)

student (firstname = 'Geeks' , lastname = 'Practise')
student( lastname = 'Practice', firstname = 'Geeks')
```

Output :-

```
Geeks Practise
Geeks Practise
```

Positional Arguments

We used the position argument during the function call so that the first argument (or value) is assigned to name and second argument is assigned to age.

ex 53.py

```
def nameAge(name, age):
    print('Hi I am :', name)
    print('My age is:', age)

print("case-1:")
nameAge("Hemu", 18)

print("case-2")
nameAge(18, "hemu")
```

Output :

```
case-1
Hi I am Hemu
My age is 18
case-2
Hi I am 18
My age is Hemu
```

Arbitrary Keyword Arguments

In python Arbitrary keyword arguments, *args and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols

- *args in Python (Non-keyword arguments)
- **kwargs in Python (Keyword arguments)

ex: ex 54.py

```
def myFun(*argv):
    for arg in argv:
        print(arg)

myFun("Hello", "Welcome", "To", "Python Tutorial").
```

**Kwargs ex 55.py

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        Print("%s == %s" % (key, value))

myFun(first = 'Hemanth', mid = 'Sai', last = 'nag').
```

Python Function within function :-

```
def f1():
    s = 'I love Geeks for Geeks'
    def f2():
        print(s)
    f2()

f1()
```

Anonymous Functions in python :-

In Python, an anonymous function means that a function is without a name.

As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

```
def cube(x): return x*x*x

cube_v2 = lambda x: x*x*x

print(cube(7))

print(cube_v2(7))
```

Output :- 343
343.

Recursive Functions in python

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)

print(factorial(4))
```

Return statement in python

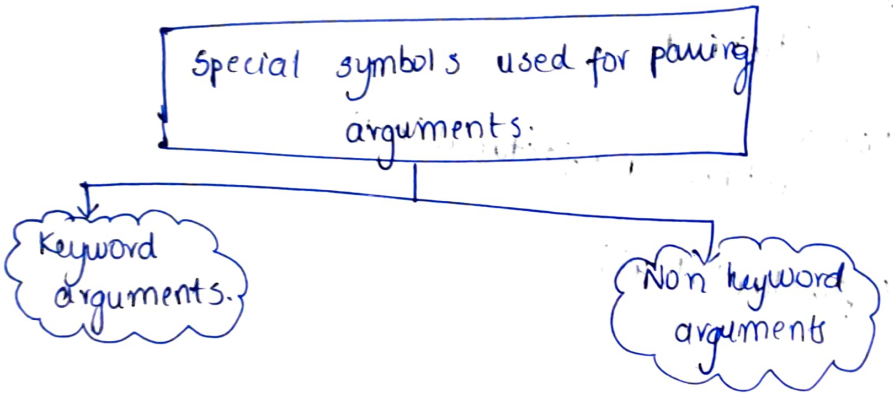
```
def square_value(num):
    return num**2

print(square_value(2))

print(square_value(2))

print(square_value(4))
```

Pass by reference:

```
def myFun(x):
    x = 20

x = 10
myFun(x)
print(x)
```

Output: 10

*args and **kwargs in python.



*args (Non-keyword arguments)

**kwargs (keyword arguments).

example args:-

```
def myFun(*argv):
    for arg in argv:
        print(arg)

myFun('Hello', 'Welcome', 'to', 'Geeks for Geeks')
```

for **kwargs

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

myFun ("i", first = "Hemanth", mid = "sai", last = "nag")
```

Using both *args and **kwargs in python to call a function

ex56.py

```
def myFun ( arg1, arg2, arg3):
    print ("arg1:", arg1)
    print ("arg2:", arg2)
    print ("arg3:", arg3)

args = ("Geeks", "for", "Geeks")
myFun (*args)
kwargs = {"arg1": "Geeks", "arg2": "for", "arg3": "Geeks"}
myFun (**kwargs).
```

ex57.py

```
def myFun(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)

myFun("Hemanth", 'sai', 'nag', first='Gnapika', mid='phani',
      last='Chilukuri');
```

Yield

When to use yield insted of return in python?

'return' sends a specified value back to its caller whereas yield can produce a sequence of values. We should use yield when we want to iterate ove a sequence but don't want to store the entire sequence in memory. yield is used in python generators.

ex58.py

```
def nextSquare():
    i = 1
    while True:
        yield i*i
        i += 1

for num in nextSquare():
    if num > 100:
        break
```

print(num).

Generators in Python :

* Generator Function in Python :-

A generator function in python is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of def contains yield, the function automatically becomes a python generator function.

ex:-

```
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

for value in simpleGeneratorFun():
    print(value).
```

Generator Object :-

→ Python Generator functions return a generator object that is iterable.

ex59.py

```
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

x = simpleGeneratorFun()

print(next(x))
print(next(x))
print(next(x))
```

Output :

```
1
2
3
```

ex60.py

Fibonacci series using generator.

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a+b

x = fib(5)
print(next(x))
print(next(x))
```

```
print (next (x))
print (next(x))
print ( next(x)).

# using for loop
for i in fib(5):
    print(i)
```

Phython Lambda :-

an anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

Syntax: lambda arguments : expression.

Example:

```
calc = lambda num : "Even number" if num%2 == 0
        else "odd number".

print( calc(20)).
```

Output: Even number.

Global and Local variable in Python :-

Python Global variable are those which are not defined inside any function and have a global scope whereas Python local variables are those which are defined inside function and their scope is limited to that function only.

Phython Closures :-

Before seeing what a closure is, we have to first understand what nested functions and non-local variable are.

Nested functions in python :

ex :-

```
def outerFunction(text):
    def innerFunction():
        print(text)
    innerFunction()

if __name__ == '__main__':
    outerFunction('Hey!')
```

Output :- Hey!

ex1.py

ex 62.py

```
def nth_power(exponent):
    def pow_of(base):
        return pow(base, exponent)
    return pow_of

squares = nth_power(2)
print(squares(2))
print(squarer(3))
print(squares(5))

cube = nth_power(3)
print(cube(2))
print(cube(3))
print(cube(4))
```

Decorators

```
def div(a,b):
    print(a/b)

def smart_div(func):
    def inner(a,b):
        if a<b:
            a,b = b,a
        return func(a,b)
    return inner

div = smart_div(div)
div(2,4)
```

Exception Handling

exception = events detected during execution that interrupt the flow of a program.

ex63.py

```
try:
    numerator = int(input("Enter a number to divide:"))
    denominator = int(input("Enter a number to divide by:"))
    result = numerator / denominator.

except ZeroDivisionError as e:
        print(e)
        print("you can't divide by zero")

except ValueError as e:
        print(e)
        print("Enter only number plz")

except Exception as e:
        print(e)
        print("something went wrong :(")

else:
        print(result)

finally:
        print("This will always execute").
```

Syntax:

```
try:
    # some code...
except:
    # optional block
    # handling of exception (if required)
else:
    # execute if no exception
finally:
    # some code..... (always executed)
```

Raise

In python, the 'raise' statement is used to raise exception explicitly.

Basic syntax of the 'raise' statement:

```
raise ExceptionType ("Optional message describing the exception")
```

ex64.py

```
def example_function(value):
    if value < 0:
        raise ValueError("Value must be non-negative")
    else:
        print("Value is:", value)

try:
    example_function(-5)
except ValueError as ve:
    print("Caught an exception:", ve)

# We will disuss user defined exceptions after OOP's
# concept
```

File detection ex65.py

```
import os

path = "c:\\"                    # use \\ to seperate directory

if os.path.exists(path):
    print("That location exists")
    if os.path.isfile(path):
        print("That is a file")
    if os.path.isdir(path):
        print("That is a folder")

else:
    print("That location doesn't exists")
```

Read a file : ex66.py

① create a file text.txt

```
try:
    with open('text.txt') as file:
        print(file.read())

    print(file.closed)

except FileNotFoundError:
    print("File was not found").
```

Python file open

- Syntax :

```
f = open (filename, mode)
```

Where the following mode is supported:

1. r : Open an existing file for a read operation.
2. w : Open an existing file for a write operation. If the file already contains some data, then it will be overriden but if the file is not present then it creates the file as well.
3. a : Open an existing file for append operations. It won't override existing data.
4. r+ : To read and write data into the file. The previous data in the file will be overriden.
5. w+ : To write and read data. It will override existing data.
6. a+ : To append and read data from the file. It won't override existing data.

→ Working in read mode:

```
file = open('text.txt','r')
for each in file
    print(each).
```

(or)

```
file = open("text.txt",'r')
print(file.read())
```

python code to illustrate read() mode character wise

```
file = open('text.txt',r)
print(file.read(5))
```

Output: Hemanth

python code to illustrate split() function.

```
with open('text2.txt','r') as file:
    data = file.readLines()
    for line in data:
        word = line.split()
        print(word).
```

working in write mode: ex 67.py

```
file = open('text2.txt', 'w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close().
```

working of append mode: ex 68.py

```
file = open('text2.txt', 'a')
file.write("this will add this line")
file.close()
```

Implementing all the functions in file handling:

ex69.py

```
import os

def create-file( filename):
    try:
        with open( filename, 'w') as f;
            f.write ('Hello, world ! \n')
        print ("File" + filename + "created succesfully.")
    except IO Error:
        print (" Error : could not create file " + filename)

def read-file (filename):
    try:
        with open ( filename, 'r') as f:
            print (f.read())
    except IOError:
        print (" Error: could not read file" + filename)

def append-file ( filename , text):
    try:
        with open ( filename , 'a') as f:
            f.write (text)
        print ("Text appended to file" + filename + "succesful")
    except IOError:
        print ("Error : Could not append to file" + filename)
```

```
def rename_file(filename, new_filename):
    try:
        os.rename(filename, new_filename)
        print("File " + filename + " renamed to " + new_filename)
    except IOError:
        print("Error: could not rename file " + filename)

def delete_file(filename):
    try:
        os.remove(filename)
        print("File " + filename + " deleted successfully")
    except IOError:
        print("Error: could not delete file " + filename)

if __name__ == '__main__':
    filename = "example.txt"
    new_filename = "new_example.txt"
    create_file(filename)
    read_file(filename)
    append_file(filename, "This is some additional text.\n")
    read_file(filename)
    rename_file(filename, new_filename)
    read_file(new_filename)
    delete_file(new_filename)
```

write:-

text

copy a file :-

copyfile() = Copies contents of a file.

copy() = copyfile() + permission mode + destination can be a directory.

copy2() = copy() = copies (file's creation and modification times.

```
import shutil.

shutil.copyfile('test.txt', 'copy.txt')  # source, dustination
```

Move a file :-

```
source = "text.txt"
destination = "C:\\Users\\Home\\Desktop\\test.txt"
try:    if os.path.exists(dustination):
except FileNotFoundError:
        print(source + "was not found")
```

Modules

menager.py

```
def hello():
    print(" Hello ! How are you? ")

def bye():
    print(" Bye ! Have a wonderful time :) ")
```

ex70.py

```
import menager
menager.hello()
menager.bye()
```

or

```
import menages as mg
mg.hello()
mg.bye()
```

(or)

```
from menager import hello, bye
hello()
bye()
```

or)

```
from menager import *
```

Object Oriented Programming in python:

class Car:

__init__ (self) is like a constructor in java and c++

example file: car.py

```
class Car :

    def __init__ (self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

    def drive(self):
        print ("This car is driving")

    def stop(self):
        print ("This car is stopped")
```

ex70.py

```
from car import Car
                ↑ object
car_1 = Car("chevy", "Corvette", 2021, "blue")

print(car_1.make)
print (car_1.model)
print (car_1.year)
print (car_1.color)
car1.drive()
car_1.stop().
```

using two object : ex71.py

Class Variables

```
class Car:

    wheels = 4   #class variable.

    def __init__(self, make, model, year, color):
        self.make = make     # instance variable
        self.model = model   #   "
        self.year = year     #   "
        sel.color = color    #   "
```

another file.

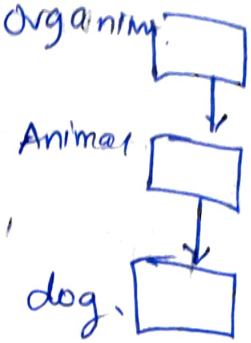
```
from car import Car

car_1 = Car("chevy", "Corvette", 2021, "blue")

car_2 = Car("Ford", "Mustang", 2022, "red")

Car.wheels = 2

print(car_1.wheels)

print(car_2.wheels)
```

Inheritance ex72.py

```
class Animal:
    alive = True
    def eat(self):
        print("This animal is eating")
    def sleep(self):
        print("This animal is sleeping")

class Rabbit(Animal):
    pass
class Fish(Animal):
    pass
class Hawk(Animal):
    pass

rabbit = Rabbit()
fish = Fish()
hawk = Hawk()

print(rabbit.alive)
fish.eat()
hawk.sleep()
```

Multi-level inheritance. ex73.py

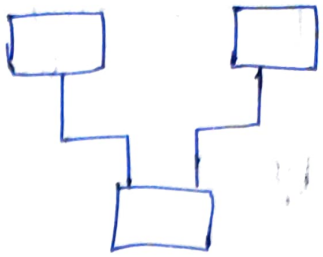
multi-level-inheritance = when a derived (child) class inherits another derived (child) class.




```
class Organism:
    alive = True

class Animal(Organism):
    def eat(self):
        print("This animal is eating")

class Dog(Animal):
    def bark(self):
        print("This dog is barking")

dog = Dog()
print(dog.alive)
dog.eat()
dog.bark()
```

~~#mu~~

Multiple - Inheritance



ex 74. py

```
class Prey:
    def flee(self):
        print("This animal flees")

class Predator:
    def hunt(self):
        print("This animal is hunting")

class Rabbit(Prey):
    pass

class Hawk(Predator):
    pass

class Fish(Prey, Predator): # multiple inheritance.
    pass

rabbit = Rabbit()
hawk = Hawk()
fish = Fish()

rabbit.flee()
hawk.hunt()
fish.flee()
fish.hunt()
```

Method Overriding:

ex75.py

```
class Animal :
    def eat(self):
        print("This animal is eating")

class Rabbit(Animal):
    def eat(self):
        print("This rabbit is eating a carrot")

rabbit = Rabbit()
rabbit.eat()
```

Method Chaining:

calling multiple method sequentially each call performs an action on the same object and returns self.

ex76.py

```
class Car:
    def turn_on(self):
        print("You start the engine")
        return self
    def drive(self):
        print("You drive the car")
        return self
```

```
    def brake (self):
        print ("you step on the brakes")
        return self.

    def turn_off(self):
        print("you turn off the engine")
        return self

car = Car()
# car.turn_on().drive()
# car.brake().turn_off()
car.turn_on().drive().brake().turn_off()
```

Super function ex77.py

super = Function used to give access to the methods of the parent class.

Returns a temporary object of a parent class when used

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width.

class Square(Rectangle):
    def __init__(self, length, width):
        super().__init__(length, width)
```

```
    def area(self):
        return self.length * self.width

Class Cube(Rectangle):
    def __init__(self, length, width, height):
        super().__init__(length, width)
        self.height = height

    def volume(self):
        return self.length * self.width * self.height

square = Square(3,3)
cube   = Cube(3,3,3)

print(square.area())
print(cube.volume())
```

Abstract class

Prevents a user from creating an object of that class

Compels a users to override abstract methods in a child class.

abstract class = a class which contains one or more abstract methods.

abstract method = a method that has a declaration but does not have an implementation.

ex78.py

```
from abc import ABC, abstractmethod.

class Vehicle(ABC):
    @abstractmethod
    def go(self):
        pass
    @abstractmethod
    def stop(self):
        pass

class Car(Vehicle):
    def go(self):
        print("you drive a car")

    def stop(self):
        print("you This car is stopped")
```

```
class Motorcycle (Vehicle):

    def go(self):
        print ("you ride a motorcycle")

    def stop(self):
        print ("this motor cycle is stopped").

car = Car()
motorcycle = Motorcycle()

car.go()
motorcycle.go()

car.stop()
motorcycle.stop().
```

Object as Argument ex79.py

```
class Car:
    color = None

def change_color (car, color)
    car.color = color

car_1 = Car()
car_2 = Car()
car_3 = Car()
```

```
print ( car_1.color)
print ( car_2. color)
print ( car_3. color)

change_ color ( car_1, "red")
change - color ( car_2, "white")
change_ color ( car_3, "blue")

print( car_1- color)
print ( car_2. color)
print ( car_3. color).
```

Duck Type :- concept where the class of an object is less important than the methods / attributes. Class type is not checked if minimum methods / attributes are present.

"If it walks like a duck, and it quacks like a duck, then it must be a duck".

ex80.py

```
class Duck :
    def walk (self):
        print (" This duck is walking")
    def talk ( self):
        print ("This duck is quacking")

class Chicken:
    def walk ( self):
        print (" This chicken is walking")
```

```
    def talk(self):
        print("This chicken is clucking")

class Person():

    def catch(self, duck)
        duck.walk()
        duck.talk()
        print("You caught the critter!")

duck = Duck()
chicken = Chicken()
person = Person()

person.catch(chicken).
```

Walrus Operator :=

walrus operator (:=)

example :

```
foods = list()
while True:
    food = input("What food do you like?:")
        if food == "quit"
        break
    foods.append(food).
```

instead of this code

with using walrus operator.

ex81.py

```
foods = list()
while food := input("What do you like?:") != "quit":
    foods.append(food)
```

Function to a variable

ex82.py

ex:

```
def hello():
    print("Hello")

hl = hello    # It assigns address.
hello()
hl()

say = print
say("Whoa! I can't believe this works").
```

Higher Order function

a function that either:

1. accepts a function as an argument

or

2. return a function.

(In python, function are also treated as objects)

lambda function:

function written in 1 line using lambda keyword

accepts any number of arguments, but only has one expression

lambda parameters : expression.

```
# def double(x):
    return x*2
print(double(5))
```

instead of this using lambda

```
double = lambda x: x*2
print(double(5))
```

ex83.py

```
double = lambda x: x*2
multiply = lambda x,y : x*y
add = lambda x,y,z : x+y+z
full_name = lambda first_name, last_name : first_name+" "+last_name.
age_check = lambda age: True if age>=18 else False.
print( double(5))
print ( multiply(5,6))
print ( add(5,6,7))
print (full_name("Hemanth " ," developer").
print (age_check (19))
```

Sort

sort() method = used with lists.

sort() function = used with iterables.

```
students = [ ("Squidward", "F", 60),
             ("sandy", "A", 33),
             ("Patrick", "D", 36),
             ("spongebob", "B", 20),
             ("Mr. krabs", "C", 78)]

age = lambda ages : ages[2]
students.sort (key = age)
for i in students:
    print(i).
```

(or)

```
sorted_students =
    sorted ( students, key = age)
for i in sorted_students:
    print(i)
```

map() = ex 85.py

applies a function to each item in an iterable (list, tuple, etc).

map (function, iterable)

ex

```
store = [("shirt", 20.00),
         ("pants", 25.00),
         ("jacket", 50.00),
         ("socks", 10.00)]

to_euros = lambda data : (data[0], data[1] * 0.82)

store_dollars = list(map(to_dollars, store))

for i in store_dollars:
    print(i).
```

filter() : creates a collection of elements from an iterable for which a function returns.

filter (function, iterable).

ex 86.py

```
friends = [("Rachel", 19),
           ("Monica", 18),
           ("Phoebe", 17),
           ("Joey", 16),
           ("Chandler", 21),
           ("Ross", 20)]

age = lambda data : data[1] >= 18
drinking_buddies =
list(filter(age, friends))

for i in drinking_buddies:
    print(i).
```

ex 87.py

reduce()

apply a function to an iterable and reduce it to a single cumulative value.

```
# reduce(function, iterable)

import functools
letters = ["H", "E", "L", "L", "O"]
word = functools.reduce(lambda x,y : x+y, letters)
print(word).
```

ex'88.py

Dictionary Comprehension:

create dictionaries using an expression can replace for loops and certain lambda functions.

```
# dictionary = { key: expression for (key, value) in iterable }

cities_in_F = {'New York': 32, 'Boston': 75, 'Los Angeles': 90,
               'Chicago': 50}

cities_in_C = { key: round ((value - 32) * (5/9)) for (key, value)
                in cities_in_F.items() }

print (cities_in_C)

desc_cities = { key: ("warm" if value >= 40 else "COLD")
                for (key, value) in cities_in_F.items() }

print (desc_cities).
```

Zip function ex 89.py

aggregate elements from two or more iterables (list, tuples, sets, etc.)

creates a zip object with paired elements stored in tuples for each element.

```
usernames = ["Dude", "Bro", "Mister"]
passwords = ("p@ssword", "abc123", "guest")

users = dict( zip( usernames, passwords))

print ( type( users))

for key, value in users.items():
    print ( key + " : " + value).
```

__name__ == "__main__"

python interpreter sets "special variables", one of which is __name__

python will assign the __name__ variable a value of '__main__' if it's the initial module being run.

Thread.

a flow of execution. Like a seperate order of instructions. However each thread takes a turn running to acheive concurrency.

GIL = (global interpreter lock), allows only one thread to hold the control of the python interpreter.

cpu bound = program / task spends most of it's time waiting for internal events (CPU)

io bound = program / task spends most of it's time waiting for external events (user input use multithreading.)

ex 90.py

```
import time
import threading

def eat_breakfast():
    time.sleep(3)
    print("you eat breakfast")

def drink_coffee():
    time.sleep(4)
    print("you drank coffee").

def study():
    time.sleep(5)
    print("you finish studying")

x = threading.Thread(target = eat_breakfast, args = ())
x.start()

y = threading.Thread(target = drink_coffee, args = ())
y.start()

z = threading.Thread(target = study, args = ())
z.start()
x.join()  y.join()  z.join()
print(threading.active_count())
print(threading.enumerate())
print(time.perf_counter())
```

daemon thread:

a thread that runs in the background, not important for program to run your program will not wait for daemon thread to complete before exiting.

non-deamon threads cannot normally be killed, stay alive until task is complete.

ex91.py

```
import threading
import time.

def timer():
    print()
    print()
    count = 0
    while True:
        time.sleep(1)
        count += 1
        print("logged in for:", count, "seconds")

n = threading.Thread(target=timer, daemon=True)
n.start()
answer = input("Do you wish to exit?")
```

Multiprocessing

running tasks in parallel on different cpu cores, bypasses GIL used for threading multiprocessing = better for cpu bound tasks (heavy cpu usage)

multithreading = better for io bound tasks (waiting around)

```
from multiprocessing import Process, cpu_count
import time.
```

ex92.py

```
def counter(num):
    count = 0
    while count < num:
        count += 1

def main():
    print(cpu_count())
    a = Process(target = counter, args = (50000000,))
    b = Process(target = counter, args = (5000000,))

    a.start()
    b.start()

    a.join()
    b.join()
    print(time.perf_counter)

if __name__ == "__main__":
    main().
```

Packages

Comands :- ① pip

② pip install --upgrade pip.

③ pip --version.

④ pip list

⑤ pip install packagename.