Examples of some Built-In Functions

Sl no	Program	Output
1	def my_func(): print("Hello World!") my_func() #calling the function	Hello World!

2	def add(a,b):
	  sum=a+b
	  print(sum)
	add(5,6)
		11

3	def add(a,b):
	  sum=a+b
	  print(sum)
		<No output>
		#There is no output because I haven't called the function

4	def add(a,b):
	  sum=a+b
	  print(sum)
	print("The sum of 5 and 6 is"), add(5,6)
		The sum of 5 and 6 is
		11

5	a=int(input("Enter 1st no:"))
	b=int(input("Enter 2nd no:"))
	def add(a,b):
	  sum=a+b
	  print(sum)
	print("The sum of", a, "and", b, "is"), add(a,b)
		Enter 1st no:4
		Enter 2nd no:5
		The sum of 4 and 5 is
		9

6	def add(a,b):
	  sum=a+b
	  print(sum)
	print(add(5,8))
		13 #Because of print(sum)
		None #Because you can't print a print function which is not returning a value

7	def name(): print("Py Prog") name()	Py Prog
8	def name(): print("Py Prog") print(name())	Py Prog None

9	def name():
	  print("Py Prog")
	  return "xyz"
	name()
		Py Prog

10	def name():
	  print("Py Prog")
	  return "xyz"
	print(name())
		Py Prog
		xyz

11	def name():
	  return "xyz"
	  print("Py Prog")
	print(name())
		xyz

12	def add(a,b):
	  sum=a+b
	  print(sum)
	add(5,6)
	print("The sum of 5 and 6 is", add(5,6))
		11 #Because of print(sum)
		11 #Because of add(5,6)
		The sum of 5 and 6 is None #Because you can't print a print function which is not returning a value

13	def sq(n): return n*n print("The square of 5 is", sq(5))	The square of 5 is 25
14	def sq(n): print(n*n) print("The square of 5 is"), sq(5)	The square of 5 is 25
15	def sq(n): print(n*n) print("The square of 5 is", sq(5))	25 The square of 5 is None
16	def name(fn,ln): print(fn,ln) print("Hello name(S,G) How are you?")	Hello name(S,G) How are you?
17	def name(fn,ln): print(fn,ln) print("Hello name("S","G") How are you?")	Invalid syntax #Because of quotes within quotes which are read as strings
18	def name(fn,ln): print(fn,ln) print("Hello", name("S","G"), "How are you?")	S G Hello None How are you?

19	def name(fn,ln): print(fn,ln) print("Hello"), name("S","G"), print("How are you?")	Hello S G How are you?
20	def name(fn,ln): return fn+ln print("Hello",name("Python","Programmer"),"How are you?")	Hello PythonProgrammer How are you?
21	def name(fn,ln): return fn+""+ln print("Hello",name("Python","Programmer"),"How are you?")	Hello PythonProgrammer How are you?
22	def name(fn,ln): return fn+" "+ln print("Hello",name("Python","Programmer"),"How are you?")	Hello Python Programmer How are you?
23	def name(fn,mn,ln): print("Hello,",fn,mn,ln) name("A","B","C")	Hello, A B C
24	def name(fn,mn,ln): return fn,mn,ln print("Hello",name("A","B","C"))	Hello ('A', 'B', 'C')
25	def name(fn,mn,ln): return fn+" "+mn+" "+ln print("Hello",name("A","B","C"))	Hello A B C

26	def name(fn,mn="B",ln="C"): #The values of mn and ln are now fixed. They are default parameters return fn+" "+mn+" "+ln print("Hello",name("A","B","C"))	Hello A B C
27	def name(fn,mn="B",ln="C"): print("Hello",fn,mn,ln) name("A")	Hello A B C
28	def name(fn,mn,ln): print("Hello",fn,mn,ln) name(mn="B",fn="A",ln="C") #Ordering doesn't matter as long as parameters have valid values	Hello A B C
29	def name(fn,mn,ln): print("Hello",fn,mn,ln) name("A","B")	TypeError: name() missing 1 required positional argument: 'ln'
30	def name(fn,mn,ln): return "Hello"+" "+fn+" "+mn+" "+ln print(name("A","B","C"))	Hello A B C

Variable Length Arguments:

Sometimes we may need to pass more arguments than those defined in the actual function.

This can be done using variable length arguments using two ways:

i) Arbitrary arguments

ii) Keyword arbitrary arguments

i) **Arbitrary arguments:** While creating a function pass '*' before the parameter name while defining the function. The function accesses the arguments by processing them in the form of a tuple. E.g.:

```
def name(*names):
  print("Hello",names[0],names[1],names[2],names[3])
name("A","B","C","D")
```

Output: Hello A B C D

ii) **Keyword arbitrary arguments:** While creating a function pass '**' before the parameter name while defining the function. The function accesses the arguments by processing them in the form of a dictionary. E.g.:

```
def name(**names):
  print("Hello",names["n1"],names["n2"],names["n3"],names["n4"])
name(n1="A",n2="B",n3="C",n4="D")
```

Output: Hello A B C D

Assert Keyword/Statement

The assert statement in Python is a debugging tool that allows us to verify assumptions about our code.
Since Python is an interpreted language, an Assert should be placed somewhere in between our code where there is a chance of a wrong user input.
When Python runs the Assert statement, it will verify the user input. If the Assert condition is True, then Python will proceed with the rest of the program. If the Assert condition is False, then Python will display an AssertionError and stop then and there.

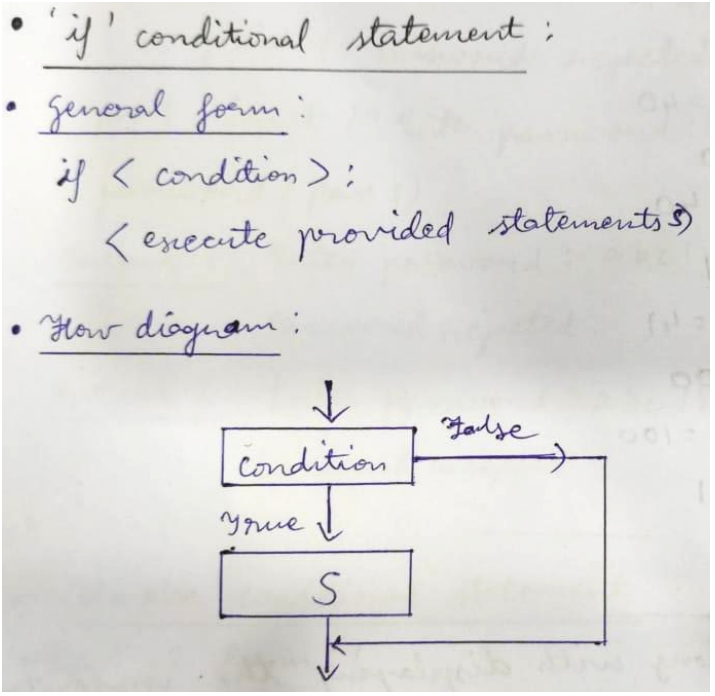
#Without assert
salary=int(input("Enter salary:"))
if salary<100:
  print("I didn't choose gareebi. Gareebi chose me.")
else:
  print("Main paisa paisa karta hu, paise pe main marta hu")
	Enter salary:-8
	I didn't choose gareebi. Gareebi chose me.
	
	Although we are getting an output, negative salary does not make sense.

salary=int(input("Enter salary:"))
assert salary>=0
if salary<100:
  print("I didn't choose gareebi. Gareebi chose me.")
else:
  print("Main paisa paisa karta hu, paise pe main marta hu")
	Enter salary:-8
	Traceback (most recent call last):
	  File
	"C:\Users\ITER\OneDrive\Desktop\1.py", line 316, in <module>
	  assert salary>=0
	AssertionError

salary=int(input("Enter salary:"))
assert salary>=0, "Salary must be >=0, because food isn't free."
if salary<100:
  print("I didn't choose gareebi. Gareebi chose me.")
else:
  print("Main paisa paisa karta hu, paise pe main marta hu")
	Enter salary:-8
	Traceback (most recent call last):
	  File
	"C:\Users\ITER\OneDrive\Desktop\1.py", line 316, in <module>
	  assert salary>=0, "Salary must be >=0, because food isn't free."
	AssertionError: Salary must be >=0, because food isn't free.

Control Structures

‘if’ conditional statement:



Q1) WAP (Write a Program) to take user input to check whether it is raining or not. Provide different outputs depending on the user input.

```
rain=input("Is it raining? Enter 'y' or 'n':")
assert rain=="y" or rain=="n"
if rain=="y":
    print("Aj mausam bada beimaan hai.")
if rain=="n":
    print("Suhana safar aur ye mausam haseen.")
```

```
Is it raining? Enter 'y' or 'n':y
Aj mausam bada beimaan hai.
==================
Is it raining? Enter 'y' or 'n':n
Suhana safar aur ye mausam haseen.
==================
Is it raining? Enter 'y' or 'n':Y
AssertionError
```

Q2) In an exam of 100 marks, 40 is the passing mark. WAP to display the student marks such that if anyone gets 38 or 39, they are upgraded to 40 and passed. Also display whether the student passed or failed.

```
def mmarks(marks,passm):
    if marks==passm-1 or marks==passm-2:
        marks=passm
    return marks
marks=int(input("Enter Marks:"))
assert marks>=0 and marks<=100
```

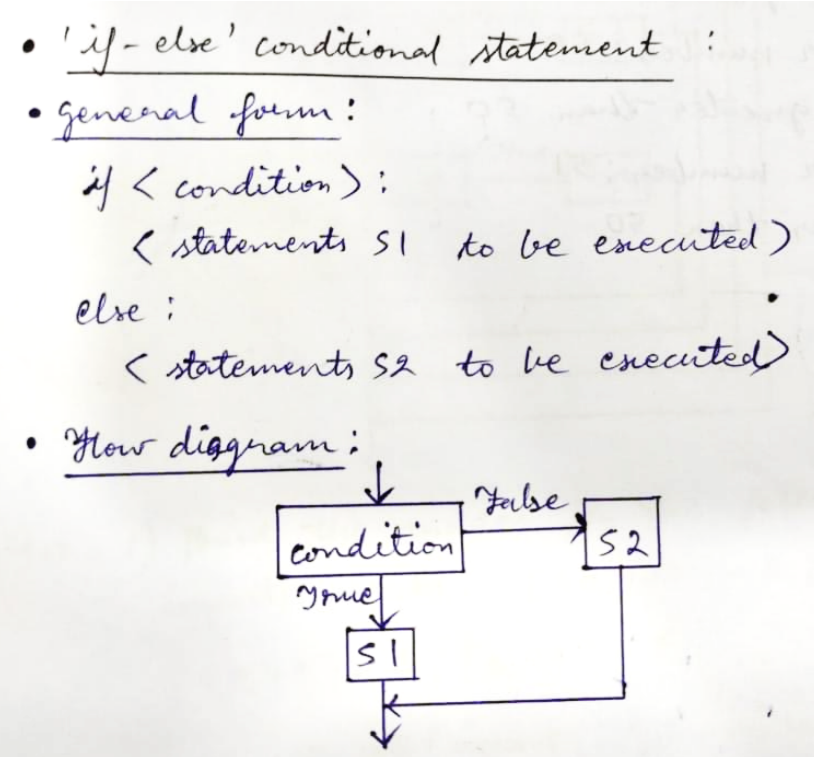
```
Enter Marks:38
Marks obtained: 40
Pass
==================
Enter Marks:39
Marks obtained: 40
```

passm=40
moderatedmarks=mmarks(marks,passm)
print("Marks obtained:",moderatedmarks)
if moderatedmarks>=40:
  print("Pass")
else:
  print("Fail")
	Pass
	==========
	Enter Marks:50
	Marks obtained: 50
	Pass
	==========
	Enter Marks:10
	Marks obtained: 10
	Fail
	==========
	Enter Marks:-8
	AssertionError
	==========
	Enter Marks:200
	AssertionError

Q3) WAP to insert the correct password (which the system knows to be “abc123”).
HW for people who have some expertise in Python (or other programming languages):
You can try to refine this program further, as mentioned below:
WAP to insert the correct password (which the system knows to be “abc123”). The user gets 3 chances. If the user inputs the password correctly in 3 chances, Python allows the user to enter. If the user fails to guess correctly in 3 chances, Python tells the user to try again after 10 minutes.

def password(passw):
  if passw=="abc123":
  print("You may enter.")
  if passw!="abc123":
  print("Bhag yaha se")
passGuess=input("Enter password:")
password(passGuess)
	Enter password:abc123
	You may enter.
	==========
	Enter password:Abc123
	Bhag yaha se

‘if-else’ conditional statement:



Q1) WAP to take user input to check whether it is raining or not. Provide different outputs depending on the user input.

rain=input("Is it raining? Enter 'y' for yes, anything else for no:")
if rain=="y":
  print("Aj mausam bada beimaan hai.")
else:
  print("Suhana safar aur ye mausam haseen.")
	Is it raining? Enter 'y' for yes, anything else for no:ff
	Suhana safar aur ye mausam haseen.
	==========
	Is it raining? Enter 'y' for yes, anything else for no:Y
	Suhana safar aur ye mausam haseen.
	==========
	Is it raining? Enter 'y' for yes, anything else for no:y
	Aj mausam bada beimaan hai.

Q2) WAP to insert the correct password (which the system knows to be “abc123”).

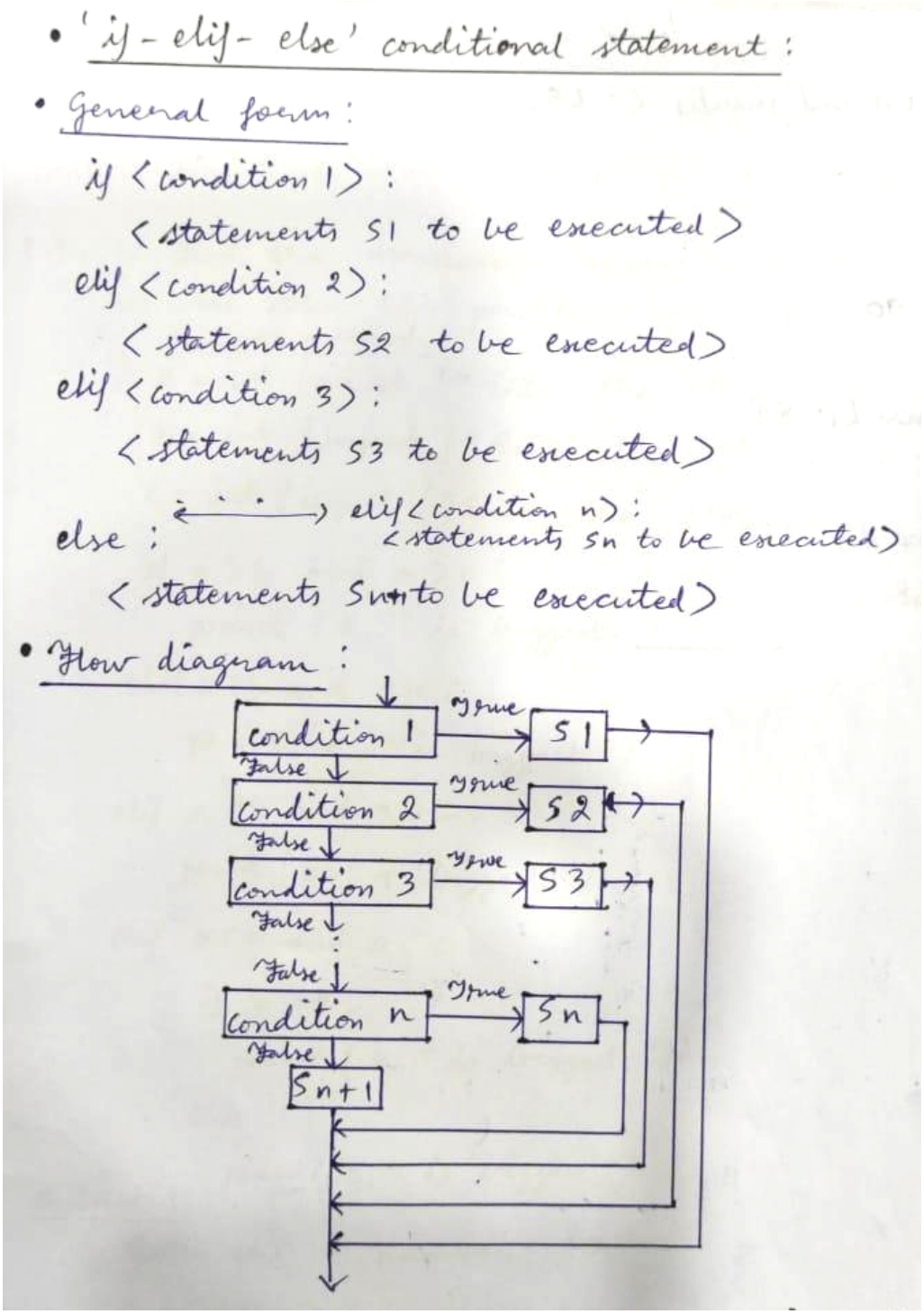
def password(passw):
  if passw=="abc123":
  print("You may enter.")
  else:
  print("Bhag yaha se")
passGuess=input("Enter password:")
password(passGuess)
	Enter password:abc123
	You may enter.
	==========
	Enter password:fdg
	Bhag yaha se

Q3) WAP to check if a user input number is greater than 50 or not.

n=int(input("Enter number:"))
if n>50:
  print(n,"is greater than 50.")
else:
  print(n,"is not greater than 50.")
	Enter number:50
	50 is not greater than 50.
	==========
	Enter number:40
	40 is not greater than 50.
	==========
	Enter number:60
	60 is greater than 50.

‘if-elif-else’ conditional statement:

Used when we have multiple if statements.

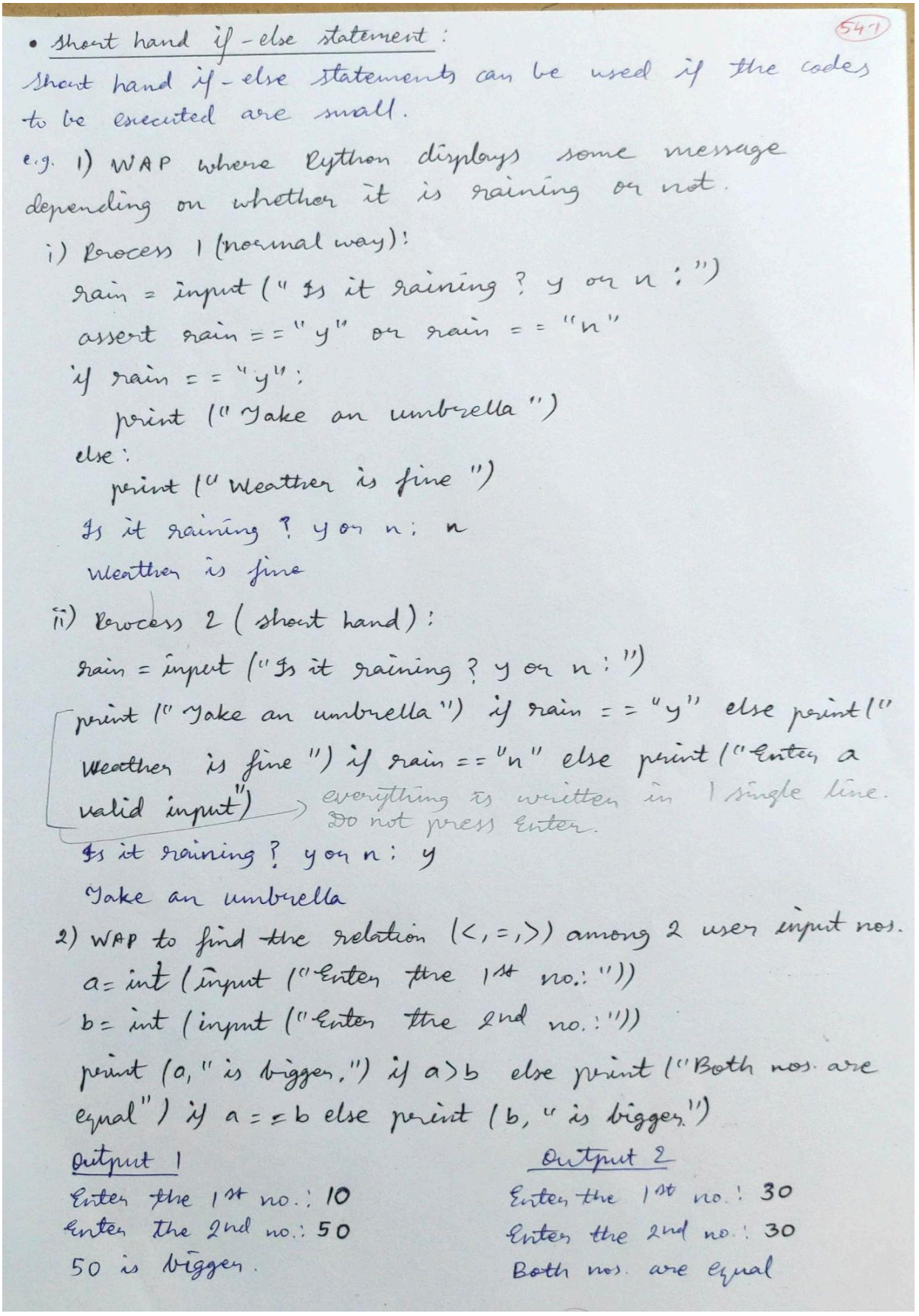


Q1) WAP to print grades for user input marks following the mentioned grading scheme:
[90, 100] - A
[80, 89] - B
[70, 79] - C
[60, 69] - D

[50, 59] - E
<59 - F

n=int(input("Enter marks:"))
if n>=90 and n<=100:
  print("Marks:",n,"Grade A.")
elif n>=80 and n<=89:
  print("Marks:",n,"Grade B.")
elif n>=70 and n<=79:
  print("Marks:",n,"Grade C.")
elif n>=60 and n<=69:
  print("Marks:",n,"Grade D.")
elif n>=50 and n<=59:
  print("Marks:",n,"Grade E.")
else:
  print("Marks:",n,"Grade F.")
	Enter marks:89
	Marks: 89 Grade B.
	==========
	Enter marks:98
	Marks: 98 Grade A.
	==========
	Enter marks:59
	Marks: 59 Grade E.
	==========
	Enter marks:20
	Marks: 20 Grade F.

Short-hand if-else statement:



• Short hand if-else statement :

Short hand if-else statements can be used if the codes to be executed are small.

e.g. 1) WAP where Python displays some message depending on whether it is raining or not.

i) Process 1 (normal way):

```
rain = input("Is it raining ? y or n : ")
assert rain == "y" or rain == "n"
if rain == "y":
    print ("Take an umbrella")
else:
    print ("Weather is fine")
```

Is it raining ? y or n : n

Weather is fine

ii) Process 2 (short hand):

```
rain = input("Is it raining ? y or n : ")
print ("Take an umbrella") if rain == "y" else print ("Weather is fine") if rain == "n" else print ("Enter a valid input")
```

→ everything is written in 1 single line. Do not press Enter.

Is it raining ? y or n : y

Take an umbrella

2) WAP to find the relation (<, =, >) among 2 user input nos.

```
a = int (input ("Enter the 1st no.: "))
b = int (input ("Enter the 2nd no.: "))
print (a, " is bigger.") if a>b else print ("Both nos. are equal") if a==b else print (b, " is bigger.")
```

Output 1

Enter the 1st no.: 10
Enter the 2nd no.: 50
50 is bigger.

Output 2

Enter the 1st no.: 30
Enter the 2nd no.: 30
Both nos. are equal

For loop:

(59) 15

• For loop :

The control statement For loop is used when we want to execute a sequence of statements a fixed number of times.

• general form:

for variable in sequence :
    <block s of statements>

• For loops can iterate over a sequence of iterable objects in Python. Iterating over a sequence is nothing but iterating over strings, lists, tuples, sets, and dictionaries.

e.g., 1) Write a program in Python to show iteration over a string.

```
name1 = "Sayantan"
for i in name1:
    print(i)
```

```
S
a
y
a
n
t
a
n
```

• Else with For loop :

• Syntax: for (condition):
        <statements in for loop>
    else :
        <statements in else>

e.g., 1)
```
for i in range(5):
    print(i)
else:
    print("no i")
```
```
0
1
2
3
4
no i
```

2)
```
for i in range(6):
    print(i)
    if i==4:
        break
else:
    print("no i")
```
```
0
1
2
3
4
```

Note 'else' is not executed. If 'else' is executed, that means the loop has executed successfully & did not break. It is not the case here.

2) Write a program in Python to show iteration over a list

```
list1 = ["A", "B", "C", "D"], ["E", "F"]]
for i in list1:
    print(i)
```

```
A
B
C
D
['E', 'F']
```

similarly, we can use loops for lists, sets, and dictionaries.

Range:

(60) • Range :

Sometimes, rather than iterating over a sequence, we want to use the for loop a specific number of times.

We use Range in this case.

• Syntax :

i) range (n) → produces a sequence of numbers from 0 to (n-1).

ii) range (1, n+1) → produces a sequence of numbers from 1 to n.

iii) range (start, end, increment) → produces a sequence of numbers from 'start' to 'end - 1' with an increment of 'increment'.

If 'increment' is not specified, the default value is taken as 1. For positive increment, we move from left to right. For negative increment, we move from right to left.

If 'start' is not specified, the default value is taken as 0.

'start', 'end', and 'increment' should be of integer type, as any other type will result in an error.

e.g., 1) for k in range (5):
    print k

0
1
2
3
4

2) for k in range (5,9):
    print k

5
6
7
8

3) for k in range (1,11,2):
    print k

1
3
5
7
9

4) for k in range (1,12,2):
    print k

1
3
5
7
9
11

5) colors = ["Red", "green", "Blue"]
for xyz in colors:
    print xyz
for i in xyz:
    print i

Red
Green
Blue
B
l
u
e

6) colors = ["Red", "Green", "Blue"]
for xyz in colors:
    print xyz
    for i in xyz:
        print i

(no space)

Red
R
e
d
Green
G
r
e
e
n
Blue
B
l
u
e

While loop:

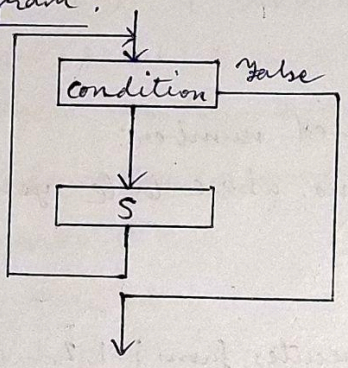
(64) • While loop :

As the name suggests, while loop executes statements while the condition is true. As soon as the condition becomes false, the interpreter comes out of the while loop.

• general form :

```
while <condition> :
    <block S of statements>
```

• Flow diagram :



e.g., 1) count = 5

```
while count >= 0 :
    count = count - 1
    print (count)
```

5 >= 0 ; 4 → count = count - 1 = 5 - 1 = 4 ; print (count) = print (4) = 4
4 >= 0 ; 3 → count = count - 1 = 4 - 1 = 3 ; print (count) = print (3) = 3
3 >= 0 ; 2 → count = count - 1 = 3 - 1 = 2 ; print (count) = print (2) = 2
2 >= 0 ; 1 → count = count - 1 = 2 - 1 = 1 ; print (count) = print (1) = 1
1 >= 0 ; 0 → count = count - 1 = 1 - 1 = 0 ; print (count) = print (0) = 0
0 >= 0 ; -1 → count = count - 1 = 0 - 1 = -1 ; print (count) = print (-1) = -1

2) count = 5

```
while count >= 0 :
    print (count)
    count = count - 1
```

5
4
3
2
1
0

3) count = 5

```
while count > 0 :
    print (count)
    count = count - 1
```

5
4
3
2
1

Else with While:

(66) • Else with While loop:

We can use else statement with the while loop. When the while loop condition becomes False, the interpreter comes out of the while loop and executes the else statement.

e.g., 1)
```
count = 5
while (count > 0):
    print (count)
    count - = 1
else:
    print ("I am inside else!")
```
5
4
3
2
1
I am inside else.

2)
```
i = 0
while i < 6:
    print (i)
    i += 1
else:
    print ("no i")
```
0
1
2
3
4
5
no i

3)
```
i = 0
while i < 6:
    print (i)
    i += 1
    if i == 4:
        break
else:
    print ("no i")
```
0
1
2
3

4)
```
i = 0
while i < 6:
    print (i)
    i += 1
    if i == 4:
        break
else:
    print ("out")
print ("Done")
```
0
1
2
3
Done

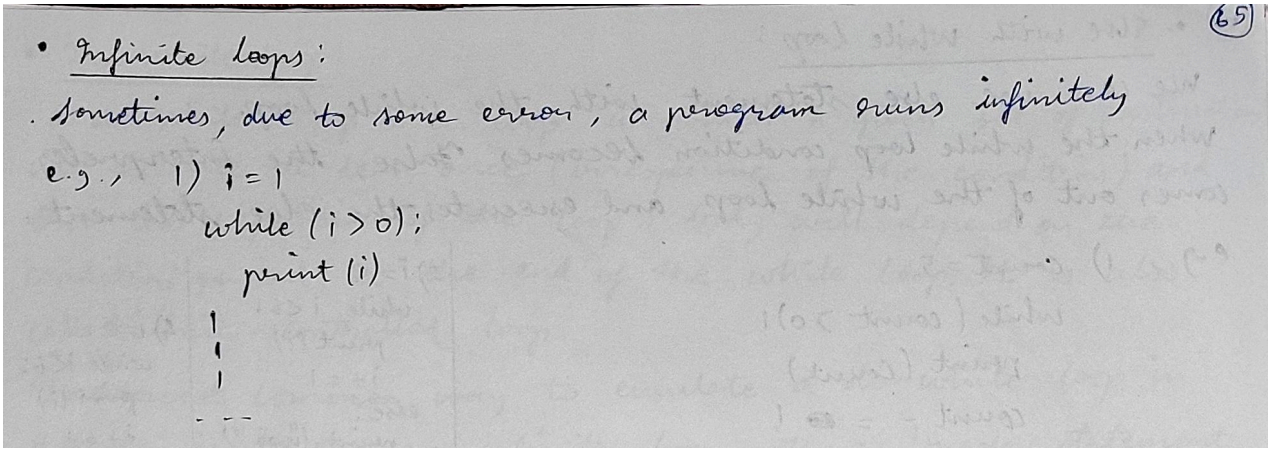
e.g., 1) Write a program in Python in which the user will be asked to input numbers as long as their number is not ≥ 40. If the number is ≥ 40, show 'Done with the loop'.

```
no1 = int (input ("Enter no.:"))
print (no1)
while (no1 < 40):
    no1 = int (input ("Enter no.:"))
    print (no1)
print ("Done with the loop")
```

↓ These types of problems are solved very easily using while. It is possible that they cannot be solved using other loops.

Enter no.: 10
10
Enter no.: 20
20
Enter no.: 15
15
Enter no.: 45
45
Done with the loop

Infinite loops:



While vs For:

Command line arguments

(50·1)

• Command line Arguments :

We know how to run programs using IDLE. Here we will learn how to run programs from the command line interface.

[To access it, go to the search menu, type cmd and press enter].

Assume we have created a script file called 1.py and saved it on the desktop.

'1.py' :
```
a = int (input ("Enter 1st no :"))
b = int (input ("Enter 2nd no :"))
print (a+b)
```

Output on IDLE:
```
Enter 1st no : 7
Enter 2nd no : 4
11
```

What if we want to run 1.py on the command prompt without opening 1.py ?

We need to make some modifications to 1.py to make it run on the command prompt.

New '1.py' :
```
import sys
print (sys.argv) → [Running this line is not essential. I have run it to show the output. One can apply # before it to not run it.]
a = int (sys.argv[1])
b = int (sys.argv[2])
print (a+b)
```

Now 1.py will not run on IDLE. There will be an IndexError : list index out of range. But it will run on command prompt. So open it. My cmd looks like this :

```
C:\Users\dell>
```

I need to open 1.py which is saved on the desktop. So I do the following :

```
C:\Users\dell>cd OneDrive\Desktop [press Enter]
```
[I type this command to reach the Desktop where 1.py is saved. You put your own directory.]

```
C:\Users\dell\OneDrive\Desktop>
```

Now I type the following :

```
C:\Users\dell\OneDrive\Desktop>python 1.py 4 7 [Enter]
```

(50·2) ['1.py', '4', '7']

||

C:\Users\dell\OneDrive\Desktop>

So we have our output || without opening the file.

How did we get it?

Observe New 1.py :

We have the following:

```
print (sys.argv)
a = int (sys.argv[1])
b = int (sys.argv[2])
```

The entire program has only 2 variables a & b. In the console when I type python 1.py 4 7 & press enter, the output is ['1.py', '4', '7']

||

Now, python 1.py 4 7 → This means I am opening the python file 1.py and within it I am taking a=4 & b=7. Due to print (sys.argv), python gives the output ['1.py', '4', '7'] in the form of a list.

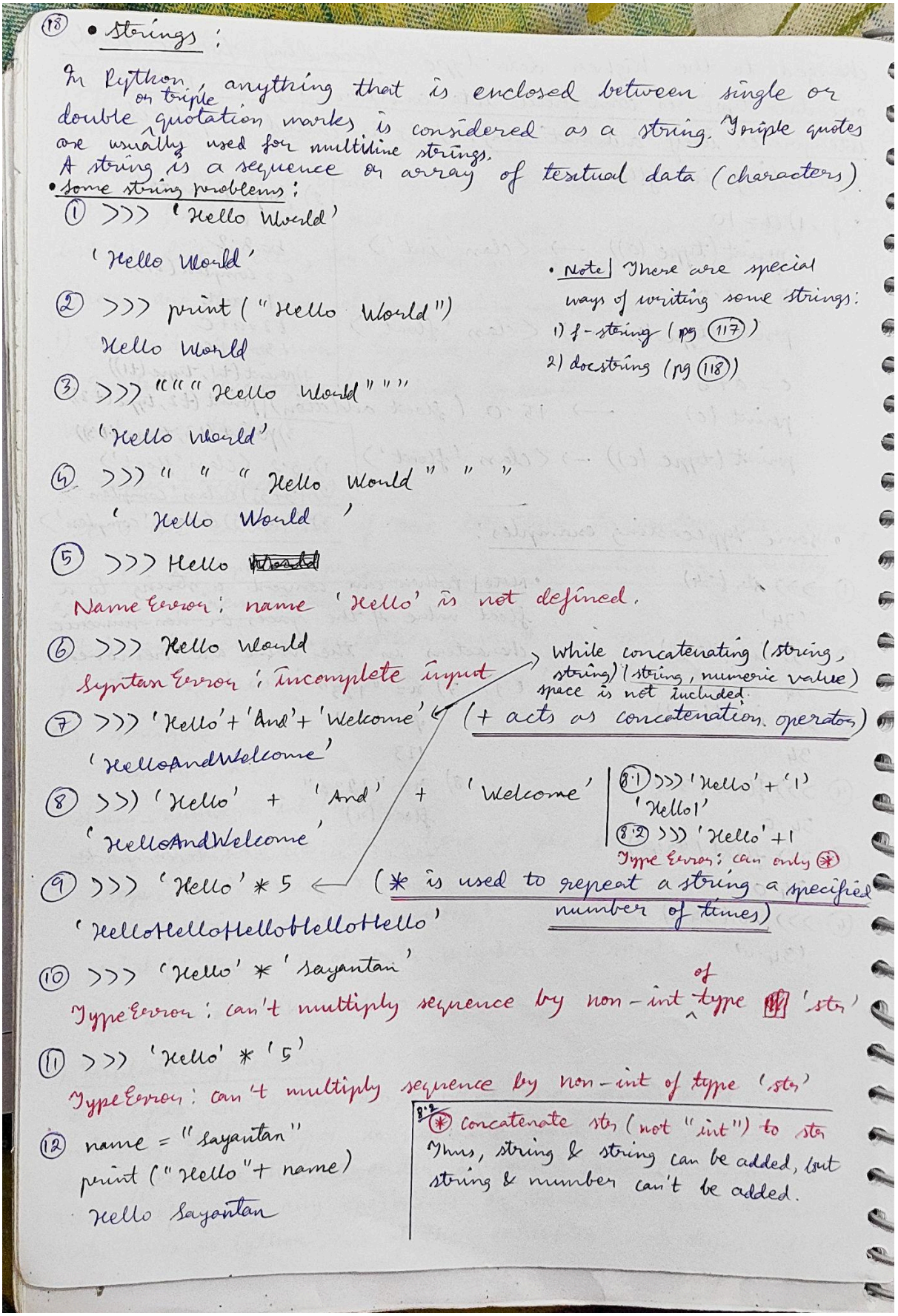
The index of element a (whose value is 4) is 1 & the index of element b (whose value is 7) is 2.

So in new 1.py we wrote a = int (sys.argv[1])

b = int (sys.argv[2])

Finally we get the output || because of print (a+b).

Strings



(18) • Strings :

In Python, anything that is enclosed between single or double or triple quotation marks is considered as a string. Triple quotes are usually used for multiline strings.
A string is a sequence or array of textual data (characters).

• Some string problems :

(1) >>> 'Hello World'
'Hello World'

(2) >>> print ("Hello World")
Hello World

(3) >>> """Hello World"""
'Hello World'

(4) >>> " " " Hello World " " "
' Hello World '

(5) >>> Hello World
NameError: name 'Hello' is not defined.

(6) >>> Hello World
SyntaxError: incomplete input

(7) >>> 'Hello' + 'And' + 'Welcome'
'HelloAndWelcome'

(8) >>> 'Hello' + 'And' + 'Welcome'
'HelloAndWelcome'

(9) >>> 'Hello' * 5
'HelloHelloHelloHelloHello'

(10) >>> 'Hello' * 'sayantan'
TypeError: can't multiply sequence by non-int of type 'str'

(11) >>> 'Hello' * '5'
TypeError: can't multiply sequence by non-int of type 'str'

(12) name = "sayantan"
print ("Hello" + name)
Hello sayantan

• Note | There are special ways of writing some strings:
1) f-string (pg (117))
2) docstring (pg (118))

While concatenating (string, string) / (string, numeric value) space is not included.

(+ acts as concatenation operator)

8.1 >>> 'Hello' + '1'
'Hello1'

8.2 >>> 'Hello' + 1
TypeError: can only (*)

(* is used to repeat a string a specified number of times)

(*) concatenate str (not "int") to str
Thus, string & string can be added, but string & number can't be added.

(13) name = "Sayantan"
print ("Hello", name)
Hello Sayantan

(14) print ('He said, "Let us study Python !" ')
He said, "Let us study Python !"

(15) poem = """Because I could not stop for Death,
He kindly stopped for me,
The carriage held but just ourselves,
And Immortality""" → an example of multiline strings
print (poem)
Because I could not stop for Death,
He kindly stopped for me,
The carriage held but just ourselves,
and Immortality.

(16) poem = """ \" Line 1,
Line 2,
Line 3. \"
- Author name """

" Line 1,
Line 2,
Line 3. "
- Author name

(17) poem = " Line 1 \n Line 2 "
print (poem)
Line 1
Line 2

(18) name = " Sayantan "
print (name[3])
name[3] = "A"
print (name)
a
TypeError: 'str' object does not support item assignment.

Strings are immutable objects (once assigned, they cannot be changed)

(19)
x = 1
1) print (x)
x_str = str (x)
2) print ("Fav no is", x, ".",
"x =", x)
3) print ("Fav no is " + x_str + "."
+ "x = " + x_str)
1) 1
2) Fav no is 1 . x = 1
3) Fav no is 1. x = 1

(20) test = input ("Enter your test :")
print (test * 5) (input
integer = int (input ("Enter no:")
print (5 * integer)
Enter your test : hi
hihihihihi
Enter no : 6
30

(20) • Accessing the characters of a string :

In Python, a string is like an array of characters.
The index of a string starts from 0 & extends to (length - 1).
Any part of a string can be accessed by using indices.
Square brackets are used to access the elements of a string,
e.g., name = "Sayantan"

S	a	y	a	n	t	a	n
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

→ length = 8
index : 0 to 7
0 to (8-1)
0 to (length - 1)

Negative indices can also be used :
index -i corresponds to the (length - i) index of i.

① name = "Raj"
print (name) ⟶ Raj
print (name[0]) ⟶ R
print (name[1]) ⟶ a
print (name[2]) ⟶ j

② name = "Raj"
print (name) ⟶ Raj
print (name[0]) ⟶ R
print (name[1]) ⟶ a
print (name[2]) ⟶ j
print (name[3]) ⟶ IndexError : string index out of range

③ name = "Raj"
for i in name :
  print (i)

} Looping through a string.

R
a
j

• String Slicing & operations on string:

A string in Python can be sliced to get a part of the string. To do this, first we need to know the indices and length of the string.

A string is essentially a sequence of characters, also called an array. Thus, we can access the elements of this array.

• Finding out the length of a string is possible using the ~~long~~ len() function.

e.g., name = "Sayantan"
print ("Sayantan is a", len(name), "letter word.")
Sayantan is a 8 letter word.

• general syntax of slicing a string with variable name 'name':

name [starting index : ending index]

↙ including starting index

↘ excluding ending index, i.e., including up to (ending−1)th index

• Slicing examples:

name = "Black RedGreen Blue" ⟶

B	l	a	c	k	␣	R	e	d	G	r	e	e	n	␣	B	l	u	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

spaces (␣) are also indexed

namelen = len(name) ⟶ 19

print (namelen) ⟶ 19

print (name [0:30]) → Black RedGreen Blue

print (name [30:]) → no output

print (name [0:10]) ⟶ Black␣RedG (∵ we have name [0;10], we start from index 0 & end in index 10−1=9)

print (name [1:9]) ⟶ lack␣Red

print (name [5]) ⟶ ␣ (Blank space which sits at index 5)

print (name [6]) ⟶ R (character sitting at index 6)

print (name [:6]) ⟶ Black␣ (slicing from the start) if nothing is mentioned in the start index place)

print (name [0:-3]) ⟶ Black␣RedGreen␣B (whenever we have −i, (*1)

print (name [:len(name)-3]) ⟶ Black␣RedGreen␣B (we have [0:16])

print (name [-1:len(name)-3]) ⟶ no output ([-1:16] = [18:16] → doesn't make sense)

print (name [-1:-3]) ⟶ no output ([-1:-3] = [18:16] → doesn't make sense)

print (name [-3:-1]) ⟶ lu ([-3:-1] = [16:18] → so we count indices 16 & 17)

print (name [10:]) ⟶ reen␣Blue (We count from index 10 to last as no end index is mentioned)

(*1) we replace it by (length − i). ∴ [0:-3] = [0:19-3] = [0:16]. So we count from 0 to 15.

More on pg (21.1) onwards.

• String methods (some of them):

• upper()

The upper method converts a string to upper case.

e.g. 1) str1 = "AabBCCddeEffF"
print(str1.upper())
AABBCCDDEEFF

• capitalize()

The capitalize method turns only the first character of the string to upper case & all the other remaining characters to lower case. The string is unaffected if the first character is already in upper case.

e.g., 1) str1 = "hello"
print(str1.capitalize())
Hello

2) str1 = "AabBCCddeEffF"
print(str1.capitalize())
Aabbccddeeff

2) a = "this is python programming."
print(a[0:4].upper() + a[4:15] + a[15:].upper())
THIS is python PROGRAMMING

3) i) a = "23all" → 23all
ii) a = "all123all" → All123all
iii) a = "12all12all" → 12all12all

• lower()

The lower function converts a string to lower case.

e.g., 1) str1 = "AabBCCddeEffF"
print(str1.lower())
aabbccddeeff

[similar to these, we have swapcase() on pg (26) & title()]

• strip()

The strip method removes all white spaces (or specified characters) before & after the string.

e.g., 1) str1 = " Python Programming "
i) print(str1)
ii) print(str1.strip())
i) Python Programming
ii) Python Programming

2) str1 = "abcd45dc123ba"
i) str1.strip("ab")
ii) str1.strip("ab31")
iii) str1.strip("abcd")
iv) str1.strip("acbd312")
i) cd45dc123
ii) cd45dc12
iii) 45dc123
iv) 45

given:
1) str1 = ".....Python,.....Programming"
• use strip to print "Programming"
If str1 = ".....Python,.....Programming"
str1 = str1.split(",")
↳ ['.....Python', '.....Programming']
lst = str1[1].lstrip("Programming") → Programming

• lstrip()

The lstrip method removes all beginning characters, as specified.

e.g., 1) name = "aaaaannnd"
print(name.lstrip("a"))
nnnd
print(name.lstrip("na"))
d

2) name = " and"
print(name.lstrip(" "))
and

- Python allows us to extract a subsequence of the form start : end : inc. This subsequence will include every inc th element of the sequence in the range start to end − 1.

```
                                  10  12  14  16  18
name = "Black RedGreen Blue"  01234 5 6789|11|13|15|17|
print (name [0:18:3] → BcRGeB
print (name [0:29:9]) → BGe
```

- Note | If increment (inc) is +ve, we move from left to right.
- If inc is −ve, we move from right to left.

- More about range on pg (60).

- Membership :

Python allows us to check for membership of the individual characters or substrings in strings using 'in' operator.

e.g., 1)))) name = "Black RedGreen Blue"

"Black" in name → True

"RedGreen" in name → True

"Red Green" in name → False

we have seen

print (name [-1:-3]) → no output as name [-1:-3] = name [18:16] & we can't go back.

But print (name [-1:-3:-1]) → eu as name [-1:-3:-1] = name [18:16:-1] = name [18], name [17]

- **rstrip()**

The rstrip method removes all trailing characters.

e.g., 1) str1 = "Heelllooo"
print(str1.rstrip("o")) ; print(str1.rstrip("ol"))
Heelll Hee

- **replace()**

The replace method replaces all occurences of a string with another string.

e.g., 1) str1 = "Python Programming"
print(str1.replace("ram", "tog"))
Python Progtogming
print(str1.replace("pr", "tog") → Python Programming } case sensitive
print(str1.replace("Pr", "tog") → Python togogramming }
2) a = "a b"; print(a.replace(" ", "PY") → PYaPY PYbPY
3) a = "Kolkata Kolkata don't worry Kolkata"
print(a.replace("Kolkata", "India", 2) → India India don't worry Kolkata

(*) Partition
3) a = "Python Programming"
print(a.partition("z"))
('Python Programming', '', '')

- **split()**

The split method splits the given string at the specified instance and returns the separated strings as list items.

e.g. 1) str1 = "Python Programming"
print(str1.split(" "))
# splits the string at the white space " "
['Python', 'Programming']
print(str1.split("ra"))
['Python Prog', 'mming']
print(str1.split("P"))
['', 'ython', 'rogramming']
print(str1.split("z"))
['Python Programming']

2) str1 = "Python, Programming"
print(str1.split("P"))
['', 'ython, ', 'rogramming]

Opposite to split & partition we have join() on pg (28).

- **partition()**

The partition function divides a string into 2 parts based on a delimiter & returns a tuple comprising string before the delimiter, the delimiter itself, and the string after the delimiter.

e.g. 1) str1 = "Python, Programming"
print(str1.partition(","))
('Python', ',', 'Programming')
2) str1 = "Python, Programming"
(*) print(str1.partition("P")) → ('', 'P', 'ython Programming')

- **count()**

Helps us count the occurrences of particular characters in strings.

1) name = "Mississippi"
name.count("M") → 1
name.count("s") → 4
name.count("z") → 0

2) name = "encyclopedia"
vowels = "aciou"
vowelcount = 0
for i in vowels:
    vowelcount += name.count(i)
print(vowelcount) → 5

Some extra problems of count on pg (29)

3) name = "Encyclopedia"
<Rest are same as 2>
print(vowelcount)
↳ 4 (E is not counted as a vowel)

(24) • center()

The center method aligns the string to the center as per the parameters decided by the user.

e.g., 1) str1 = "Python Programming"

i) print(str1.center(0))

ii) print(str1.center(50))

iii) print(str1.center(100))

iv) print(str1.center())

i) Python Programming

ii) Python Programming

iii) Python Programming

iv) TypeError: center expected at least 1 argument, got 0.

2) We can also provide a padding character. It will fill the empty spaces with the character provided by the user.

str1 = "Python Programming"

i) print(str1.center(0, "."))

ii) print(str1.center(50, "#"))

iii) print(str1.center(100, "@"))

i) Python Programming

ii) ##########Python Programming##########

iii) @@@@@@@@@@@@@@@@@@@@Python Programming@@@@@@@@@@@@@@@@@@@@

• count() (already done on pg (23))

The count method returns the number of times the given value has occurred within the given string.

e.g., 1) str1 = "Mississippi Chillipines"

print(str1.count("i"), str1.count("P"), str1.count("p"), str1.count("s"), str1.count("S"))

7 1 3 5 0

2) a = "Python"

print(a.count("")) → 7 [Why? .P.y.t.h.o.n.]

- **endswith()**

The endswith method checks if the string ends with a given value. If yes, then return True, else return False.

[similar to this, we have startswith() on pg 26]

e.g., 1) str1 = "Python Programming"

```
print(str1.endswith("ing")) ⟶ True
print(str1.endswith("inG")) ⟶ False
```

We can also check for a value in between the string by providing start and end index positions.

e.g., 2) str1 = "Python Programming"
0 1 2 3 4 5 6 7 8 9 10 11 12 13

```
print(str1.endswith("ra", 4, 13)) ⟶ True
print(str1.endswith("ra", 4, 12)) ⟶ False
("og", 9, 11) ⟶ True
("og", 10, 11) ⟶ False
```

- **find()**

The find method searches for the first occurence of the given value and returns the index where it is present. If a given value is absent in the string, then return −1.

e.g., 1) str1 = "Python Programming"
0 1 2 3 4 5 6 7 8 9

```
print(str1.find("og")) ⟶ 9
print(str1.find("P")) ⟶ 0
print(str1.find("z")) ⟶ −1
print(str1.find("o")) ⟶ 4
```

- **rfind()** To find the last occurrence of a string, we use 'rfind' which scans the string in reverse order from right end to the beginning.

e.g., 1) name = "red green red"
0 1 2 3 4 5 6 7 8 9 10 11 12

```
print(name.rfind("red")) ⟶ 10
print(name.rfind("Red")) ⟶ -1
```

- **index()**

The working principle is the same as find method. The index method searches for the first occurence of the given value & returns the index where it is present. If the given value is absent in the string, then raise an exception (this is the difference between index & find).

e.g., 1) str1 = "Python Programming"

```
print(str1.index("og")) ⟶ 9
print(str1.index("P")) ⟶ 0
print(str1.index("z")) ⟶ ValueError: substring not found
```

- **startswith()**

The startswith method checks if the string starts with a given value. If yes, then return True, else return False.

e.g., 1) str1 = "Python Programming"
print(str1.startswith("Python")) → True
print(str1.startswith("python")) → False

- **swapcase()**

The swapcase method changes the character casing of the string. Upper cases are converted to lower cases & vice versa.

e.g., 1) str1 = "Python Programming 1"
print(str1.swapcase()) → pYTHON pROGRAMMING 1

- **title()**

The title method capitalizes each letter of the words in the string.

e.g., 1) str1 = "I'm doing Python Programming by myself")
print(str1.title())

I'M Doing Python Programming By Myself

3) i) a = "23all" → 23All
ii) a = "all23all" → All23All
iii) a = "12all12all" → 12All12All

2) a = "PYTHON Programming" print(a.title()) → Python Programming

- **isalnum()**

The isalnum method returns True only if the entire string consists only of A-Z, a-z, 0-9. If any other characters or punctuations are present, then it returns False.

e.g., 1) str1 = "Python Programming"
print(str1.isalnum()) → False (due to space)
2) str2 = "PythonProgramming"
print(str2.isalnum()) → True
3) str3 = "Python1"
print(str3.isalnum()) → True

- **isdigit()** checks if a string consists of only digits or not.

e.g. 1) name = "1234"
print(name.isdigit()) → True

2) name = "123 456"
print(name.isdigit()) → False

- **isalpha()**

The isalpha method returns True only if the entire string contains only A-Z, a-z. If any other characters, or punctuations, or numbers (0-9) are present, then it returns False.

e.g., 1) str1 = "Python"
str2 = "Python1"
print(str1.isalpha()) ⟶ True
print(str2.isalpha()) ⟶ False

2) Write a program to count the total nos. of numbers, letters, and special characters in a User input string.

```
s1 = input("Enter string:")
cnum, clet, cspl = 0,0,0
for i in s1:
    if i.isdigit()==True:
        cnum += 1
    elif i.isalpha()==True:
        clet += 1
    else:
        cspl += 1
print("Numbers, letters, spl chars:", cnum, clet, cspl)
```

Enter string: P@#yn26at&i5ve
Numbers, letters, spl chars:
3 8 4

- **islower()**

The islower method returns True if all characters in the string are in lower case. Otherwise it returns False.

e.g., 1) str1 = "pYthon programming"
str2 = "python Programming"
str3 = "123456"
print(str1.islower()) ⟶ True
print(str2.islower()) ⟶ False
print(str3.islower()) ⟶ False

- **isupper()**

The isupper method returns True if all characters in the string are in upper case. Otherwise it returns False.

e.g., 1) str1 = "PYTHON"
str2 = "PYthon"
str3 = "123456"
print(str1.isupper()) ⟶ True
print(str2.isupper()) ⟶ False
print(str3.isupper()) ⟶ False

- **isprintable()**

The isprintable method returns True if all characters in the string are printable. Otherwise it returns False.

e.g., 1) str1 = "Python Programming"
str2 = "Python Programming \n"
print(str1.isprintable()) ⟶ True
print(str2.isprintable()) ⟶ False

(28) • isspace()

The isspace method returns True only if the string contains white space. Else it returns False.

e.g. 1) str 1 = "Python Programming"
str 2 = " " # using spacebar
str 3 = " " # using tab
print (str 1. isspace()) → False
print (str 2. isspace()) → True
print (str 3. isspace()) → True

• istitle()

The istitle method returns True only if the first letter of each word of the string is capitalized. Else it returns False.

e.g., 1) str 1 = "Python Programming"
str 2 = "Python programming"
print (str 1. istitle()) → True
print (str 2. istitle()) → False

a1 = "I'm Doing Python"
a2 = "I'M Doing Python"
a3 = "Python Program"
print (a1. istitle()) → False
print (a2. istitle()) → True
print (a3. istitle()) → False

• join()

The join function returns a string comprising elements of a sequence of strings separated by the specified delimiter.

e.g. 1) name = "I am Sayantan"
"." . join (name)
'I . . a . m . . S . a . y . a . n . t . a . n'

2) name = ["I", "am", "Sayantan"]
";" . join (name)
'I ; am ; Sayantan'

• encode() and decode()

Sometimes we need to transform data from one format to another for the sake of compatability. We use 'encode' that returns the encoded version of a string, based on the given encoding scheme. Another function 'decode' (opposite of encode) returns the decoded string.

e.g., 1) name = "Sam"
Encodedstring = name.encode('utf 32')
print (Encodedstring) — (1)
Decodedstring = Encodedstring . decode ('utf 32')
print (Decodedstring) — (2)

(1) b'\xff\xfe\x00\x00S\x00\x00\x00a\x00\x00\x00m\x00\x00\x00'
(2) Sam

• Some string programs:

1) Write a program in Python to count the number of matching characters in a pair of strings.

```
def charmatch(str1, str2):
    lower1 = str1.lower()
    lower2 = str2.lower()
    count = 0
    for ch1 in lower1:
        for ch2 in lower2:
            if ch1 == ch2:
                count += 1
    return count
name1 = input("Enter the first string: ")
name2 = input("Enter the second string: ")
print("The number of matching characters in", name1,
      "and", name2, "is:", charmatch(name1, name2))
```

Enter the first string : Sayantan
Enter the second string : Satyameva
The number of maching characters in Sayantan and Satyameva is : 12

How it works:

lower1 = sayantan
lower2 = satyameva

Take 's' in lower1:
  Compare every character in lower2:
    's' of Lower1 matches with 1 's' of Lower 2
    count = 0+1=1
  go back

Take 'a' in lower1:
  Compare every character in Lower 2:
    'a' in lower1 matches with 3 'a'-s of lower 2
    count = 1+3=4
  go back

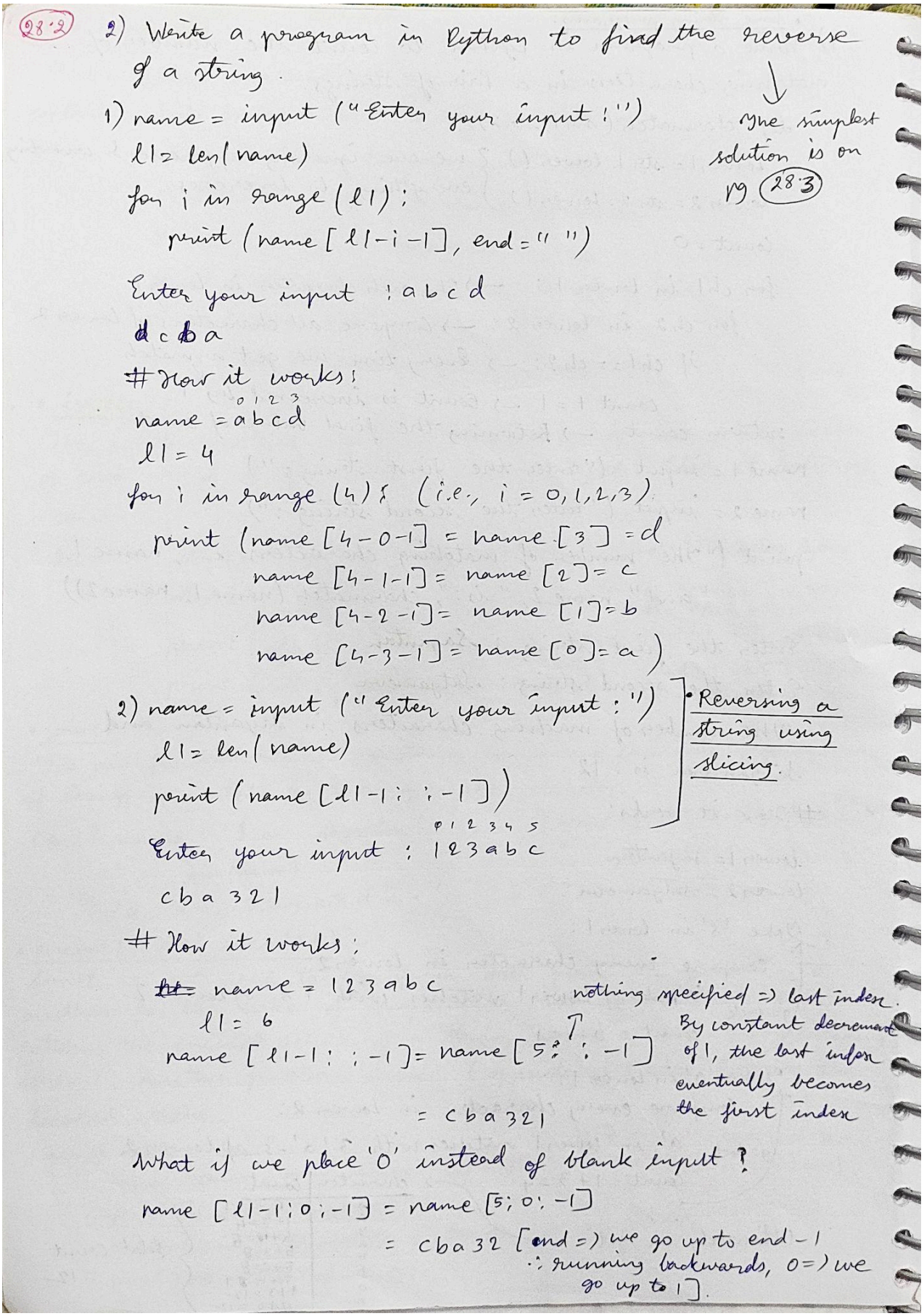
...

Ultimately we get →

character	count
s	0+1=1
a	1+3=4
y	4+1=5
a	5+3=8
n	8+0=8
t	7+1=9
a	9+3=12
n	12+0=12

total count = 12

Finding the reverse of a string:



(28.2)

2) Write a program in Python to find the reverse of a string.

The simplest solution is on pg (28.3)

1) name = input ("Enter your input :")
ll = len(name)
for i in range(ll):
    print(name[ll-i-1], end = " ")

Enter your input : a b c d
d c b a

# How it works:
name = a b c d
       0 1 2 3
ll = 4
for i in range(4): (i.e., i = 0, 1, 2, 3)
    print(name[4-0-1] = name[3] = d
          name[4-1-1] = name[2] = c
          name[4-2-1] = name[1] = b
          name[4-3-1] = name[0] = a)

2) name = input ("Enter your input :")
ll = len(name)
print(name[ll-1::-1])

• Reversing a string using slicing.

Enter your input : 1 2 3 a b c
                   0 1 2 3 4 5
c b a 3 2 1

# How it works:
name = 1 2 3 a b c
ll = 6
name[ll-1::-1] = name[5::-1]

nothing specified => last index. By constant decrement of 1, the last index eventually becomes the first index

= c b a 3 2 1

What if we place '0' instead of blank input?

name[ll-1:0:-1] = name[5:0:-1]
= c b a 3 2 [end => we go up to end - 1 ∴ running backwards, 0 => we go up to 1].

3) name = input("Name:") → Name: abc123

print(name[::-1]) → 321cba

[::-1] is the same as [-1:-(len(s)+1):-1]

Why: Range: (start : finish : inc)

If there is no start, Python assumes start = index 0.

If there is no finish, Python assumes finish = last index

If there is no inc, Python assumes inc = 1. & for a +ve inc. we move from left to right.

e.g., name = "abc123"

print(name[:]) → ~~321cba~~ abc123

suppose, inc = -1. For -ve inc we move from right to left. ∴ our movement is reversed, by default Python assumes start = last index & finish = index 0. Similar results are obtained for other -ve inc values.

e.g., name = "abc123"

print(name[::-1])

321cba

name = "abc123"

print(name[::-2])

31b

name = "abc123"

print(name[::-4])

3b

Sep and End functions

22.2 • sep & end functions:

• sep:

The sep parameter helps us print multiple values in a Python program in a readable manner. The sep parameter differentiates between the objects.

e.g., 1) print ("1", "2", "3") → 1 2 3

2) print ("1", "2", "3", sep = "") → 123

3) print ("1", "2", "3", sep = "␣") → 1 2 3 → same as (1).

4) print ("1", "2", "3", sep = ".") → 1.2.3

5) print ("1", "2", "3", sep = "\n") →
1
2
3

6) print ("1", "2", "3", sep = "\n\n") → 1\n\n2\n\n3

7) a = 1
b = 'xyz'
print (a, b, sep = ",") → 1, xyz

• end:

By default, the print statements ends with a new line. The end parameter is used to append any string at the end of the output of the print statement.

e.g., 1) print ("a") } a
print ("b") } b
What if we want to print them on the same line?

2) print ("a", end = " ") } a b
print ("b")

3) print ("a", end = "") } ab
print ("b")

4) a = 1
b = 'xyz'
print (a, b, end = ",") → 1 xyz,

5) print ("a", end = " breaks ") } a breaks b
print ("b")

6) print ("a", "b", "c", "d", sep = "_", end = "e")
a_b_c_d e

7) print ("a", "b", "c", "d", sep = "_", end = "_e")
a_b_c_d_e

Scope and Namespace

(76) • Objects and object ids:

Each object in Python is assigned a unique identifier that can be accessed using the function "id".

e.g., 1) a = 5
print ("a = ", a, " id(a): ", id(a)) —— (1)
b = 2+3
print ("b = ", b, " id(b): ", id(b)) —— (2)
a = 7
print ("a = ", a, " id(a): ", id(a)) —— (3)
print ("b = ", b, " id(b): ", id(b)) —— (4)

Global frame

a | 5

a | 5
b | 5

a | 7
b | 5

(1) a = 5 id(a): 139979450419264
(2) b = 5 id(b): 139979450419264
(3) a = 7 id(a): 139979450419328
(4) b = 5 id(b): 139979450419264

In (1), an int object 5 is created and a name "a" is assigned to it.
(2) does not create a new object. It only associates the name "b" to the int object 5 created earlier. Thus, a & b have the same object id.
(3) creates an int object 7 and assigns a name "a" to it.
In (4), b continues to be associated with int object 5 created earlier.

• Note: i) Different object ids are generated when a script is executed again.

ii) You can use the following website to execute & visualize Python programs:

http://www.pythontutor.com. Note: Python caches or interns small integer objects (typically up to 100) for future use. But the same may not hold for other forms of data, e.g.,

2) >>> print (id(1.5))
2969434280336
>>> print (id(1.5))
2969398775952
>>> print (id(1.5))
2969434280432

>>> print (id(1.5))
2969434280336
>>> print (id(1.5))
2969434280432
>>> print (id(1.5))
2969434280336

• Note: The first 3 instructions created new objects. The subsequent instructions sometimes used objects created earlier.

• del operator :

It makes a name (i.e., the association between the name and the object) undefined.

e.g., 1) >>> a = 1.5

>>> print (id(a))

2969434280432

>>> del a

>>> print (id(a))

NameError : name 'a' is not defined

Global frame

a | 1.5

2) >>> a = 5

b = 2+3

print (id(a), id(b)) ——— (1)

print (a) ——— (2)

del a

print (b) ——— (3)

del (b)

print (id(b))

Global frame

a | 5

a | 5

b | 5

b | 5

(1) 140130641266752 140130641266752

(2) 5

(3) 5

NameError : name 'b' is not defined

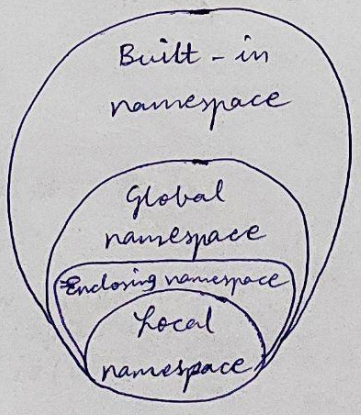
(78) • Namespaces:

- As the term suggests, a namespace is a space that holds some names.
- It defines a mapping of names to the associated objects, i.e., a namespace maps names to objects.
- In Python, a module, class, or function defines a namespace.
- Names appearing in the global frame (usually outside of the definition of classes, functions, and objects) are called global names and collectively, they define the namespace called global namespace.
- The names introduced in a class / function are said to be local to it.
- The region in a script in which a name is accessible is called its scope. Thus, the scope of a name is resolved in the content of the namespace in which it is defined.

• Types of namespaces:

- When the Python module runs solely without any user-defined modules, methods, classes, etc., some functions like print(), id(), etc., are always present. These are built-in namespaces.
- When a user creates a module, a global namespace is created. These are names defined in Python script, but usually outside of any function or class definition.
- The creation of local functions creates local namespaces.

Hierarchy:

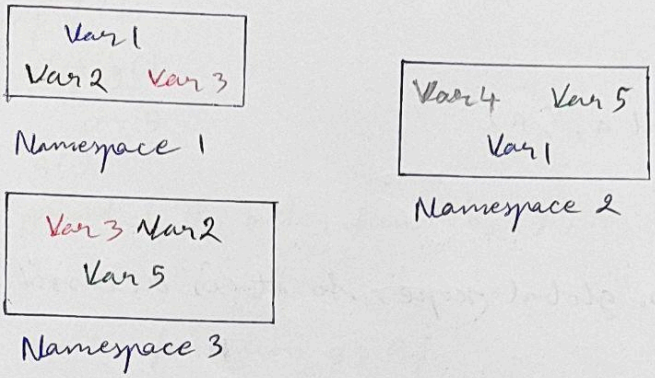


↑ LEGB rule

• Lifetime of a namespace:

- The lifetime of a namespace depends on the scope of objects.
- If the scope of an object ends, the lifetime of that object also comes to an end.
- Thus, it is not possible to access the inner namespace's objects from an outer namespace.

• Note :



As shown in the above figure, the same object name can be present in multiple namespaces as isolation between the same name is maintained by their namespace.

• Scope :

Scope refers to the coding region from which a particular Python object is accessible. Hence, one cannot access any particular object from anywhere in the code, the accessing has to be allowed by the scope of the object.

The scope rules for names in Python are often summarized as LEGB — local, Enclosing, Global, Built-in (~~title~~ the hierarchy on pg (78)).

• The scope of a ~~function~~ name in the content of a function, say f, is described as follows:

i) All names introduced locally in f can be accessed within it.

ii) If a name is not defined locally in f, but defined in a function g that encloses it, then it is accessible in f.

iii) If the name being accessed is not defined in g that immediately encloses f, then Python looks for it recursively in a function that encloses g, and so on.

iv) If a name is not defined in any enclosing function, Python looks for it outside of all functions, and if found, it is accessible in function f.

1) a = 4
```
def f():
    print("global a:", a)
f()
```
global a : 4

The variable 'a' has global scope. So it is accessible in f.

2) a = 4.5
```
def f():
    a = 5
    print("local a:", a)
f()
print("global a:", a)
```
local a : 5
global a : 4.5

'a' introduced here is local to f and has associated value 5. ∴ defining the value of name 'a' in the function f doesn't affect the value of global name 'a'.

3) a = 6
```
def f():
    a = 5
    def g():
        b = a
        print("Inside function g, b:", b)
    g()
f()
```
Inside function g, b : 5

During the execution of function g, when 'a' is to be accessed, Python looks for it locally (local scope) in g. But it is not defined in g. ∴ Python searches for it in the next enclosing scope, i.e., the scope of function f.
The variable a is defined in f.
∴ the value 5 of variable a defined in f gets bound to the occurence of variable a defined in g.

4)
```
def f():
    def g():
        a = 5
    g()
    print("In outer function g, a :", a)
f()
```

In outer function g, a:

NameError: global name 'a' is not defined

Here, 'a' is defined in the body of the inner function 'g'. When we attempt to access the name 'a', Python looks for its definition first inside the body of function f. Since there is no definition of a in f, it looks for the definition of a in the next available enclosing scope, which is the global scope. But again, there is no definition of the name a in the global name-space. Thus, the error message is shown.

5)
```
def f():
    print("Inside f")
    def g():
        var = 10
        print("Inside g, value of var :", var)
    g()
    print("Outside of g, var :", var)
f()
```

Inside f

Inside g, value of var : 10

NameError: name 'var' is not defined

6)
```
a = 4
def f():
    a = 5
    def g():
        b = a
        print("inside g:", "a =", a, "b =", b)
        a = 5
    g()
f()
```

UnboundLocalError: local variable 'a' referenced before assignment

When 'a' is accessed, Python looks for the local definition of a in g. It does find a def^{n} of a in g, but it is after where it is used. Thus, we get the error message.

7) nonlocal

In order to modify the value of a local variable in the enclosing scope, Python provides the keyword nonlocal. Using nonlocal in this program, we tell Python that the name 'a' used in g refers to the same object as the one associated with name 'a' defined in f because f is enclosing g.

```
a = 4
def f():
    a = 5
    def g():
        nonlocal a
        print("inside g, before modifying a,", "a =", a)   —(2)
        a = 10
        print("inside g, after modifying a,", "a =", a)   —(3)
    print("inside f, before calling g,", "a =", a)   —(1)
    g()
    print("inside f, after calling g,", "a =", a)   —(4)
f()
print("after calling f,", "a =", a)   —(5)
```

1) inside f, before calling g, a = 5
2) inside g, before modifying a, a = 5
3) inside g, after modifying a, a = 10
4) inside f, after calling g, a = 10
5) after calling f, a = 4

(8) global

```
01 a = 3
02 def f():
03     def g():
04         global a
05         a = 4
06         print("inside g, global a =", a)
07     g()
08     a = 5
09     print("inside f, local a =", a)
10 f()
11 print("outside all functions, a =", a)
```

~~print~~ inside g, global a = 4

inside f, local a = 5

outside all functions, a = 4

The assignment statement in line 5 modifies the global variable a, the new value being 4. In line 8, as the variable a is locally availble, Python doesn't search it in the global scope and 5 is assigned to local variable a. However, the assignment of value 5 to local variable in function f doesn't affect the value of the global variable which remains unattered, i.e., 4.

Mutable and Immutable objects in Python

- Mutable & Immutable objects in Python

Mutable → Mutation (to mutate / change).

- An object is called mutable if its state or value can be modified after it is created. Thus, we can alter its internal data / attributes without creating a new object, e.g., list, set, dictionary, user-defined classes, etc.

Examples of why list, set, and dictionary are mutable:

- list1 = [1,2,3]
  list1.append(4)
  print(list1) → [1,2,3,4]
  list1.insert(1,5)
  print(list1) → [1,5,2,3,4]
  list1.remove(2)
  print(list1) → [1,5,3,4]
  pop_element = list1.pop(0)
  print(list1) → [5,3,4]
  print(pop_element) → 1

  • More about lists on pg (85)

- dict1 = {"name": "Sayantan", "age": 31}
  new_dict = dict1
  new_dict["age"] = 37
  print(dict1) → {"name": "Sayantan", "age": 37}
  print(new_dict) → {'name': 'Sayantan', 'age': 37}

  • More about dictionaries on pg (111)

- set1 = {1,2,3}
  new_set = set1
  new_set.add(4)
  print(set1) → {1,2,3,4}
  print(new_set) → {1,2,3,4}

  • More about sets on pg (100).

- Note] The use of mutable objects is recommended when there is a need to change the size or content of the object.

(14) Immutable objects cannot be changed after they are created, e.g., numbers (integer, rational, float, decimal, complex), booleans, strings, tuples, frozen sets, user-defined classes, etc.

Examples of why strings & tuples are immutable:

- str1 = "Welcome to Python Programming"
  str1[0] = 'm'
  print(str1) → TypeError: 'str' object does not support item assignment

  • More about strings on pg (13).

- tup1 = (1, 2, 3)
  tup1[0] = 4
  print(tup1) → TypeError: 'tuple' object does not support item assignment.

  • More about tuples on pg (108).

List

- Lists
- Lists are ordered collection of data items.
- It is non-scalar type (elementary types of data such as numeric & Boolean are called scalar data types).
- A list can store any type of value, e.g., string, integer, float. Then can simultaneously store different types of values.
- List items are separated by commas & enclosed within square brackets [ ].
- Lists are mutable.

e.g., 1)
```
list 1 = [1, 2, 3, 4, 5]
list 2 = ["Red", "Green", "Blue"]
list 3 = ["Python", 1, "Programming", 2]
print (list 1)
print (type (list 1))
print (list 2)
print (list 3)
```

```
[1, 2, 3, 4, 5]
['Red', 'Green', 'Blue']
['Python', 1, 'Programming', 2]
<class 'list'>
```

List

- Name association ;

e.g., 1)
```
list 1 = [1, 2, 3, 4, 5]
print (id (list 1))
temp = list 1
print (id (temp))
print (temp)
```

```
140177810074368
140177810074368
[1, 2, 3, 4, 5]
```

Note that each of the names list 1 & temp is associated with the same list object having object id 140177810074368.

(86) • List index :

Each element in a list has its own index just like a string. The index can be used to access any particular item from the list. The 1st item has index [0], the 2nd item has index [1], the 3rd item has index [2], and so on.

e.g., 1) list 1 = [1, 2, 3, "Python", 5]

print (len (list 1))

print (list 1 [1])

print (list 1 [4])

print (list 1 [5])

index : 0 1 2 3 4
1, 2, 3, "Python", 5

5
2
5

IndexError : list index out of range

2) list 1 = [1, 2, 3, "Python", 5, 6]

list 1 [3] = 4

print (list 1)

1 2 3 4 5 6

3) list 1 = [1, 2, 3, "Python", "Apple"]

0 1 2 3 4
1, 2, 3, "Python", "Apple"

print (list 1 [-1])

print (list 1 [-3])

print (list 1 [-5])

print (list 1 [-6])

length = 5

list [-1] = list [length -1] = list [4]

list [-3] = list [2] = 3

list [-5] = list [0] = 1

list [-6] = list [-1] -> doesn't exist.

This list [-1] & the original list [-1] are different.

Apple
3
1

IndexError : list index out of range

- **List of list:**

e.g., 1) subcode = [["Physics",1], ["Chemistry",2], ["maths",3]]

print (subcode [1]) → ['Chemistry', 2]

print (subcode [1][0]) → Chemistry

print (subcode [2][1]) → 3

In general, subcode [i] refers to index i in subcode.
Subcode [i][j] refers to the j^{th} index element within the i^{th} index of subcode.

- **user input():**

Sometimes we need to take a list as an input from the user. For this we use the 'input' function and apply the 'eval' function to transform the raw string to a list:

e.g., 1) details = eval(input ("Enter course details :"))
print (details)
Enter course details : ["Python", "Easy", 1991]
['Python', 'Easy', 1991]

- **Range of index:**

- Syntax: listname [start : end : jumptoindex]

e.g., 1) letters = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
print (letters [1 : 8 : 3])
['B', 'E', 'H']
print (letters [4 :]) → ['E', 'F', 'G', 'H', 'I', 'J']
print (letters [: 4]) → ['A', 'B', 'C', 'D']
print (letters [: -3]) → same as print (letters [: 10-3])
= ['A', 'B', 'C', 'D', 'E', 'F', 'G']

print (letters [: : 2]) → ['A', 'C', 'E', 'G', 'I']
print (letters [-8 : -1 : 2]) → same as print (letters [2 : 9 : 2])
= ['C', 'E', 'G', 'I']

(88) • List Comprehension:

List comprehensions are used to create new lists from other iterables like lists, tuples, dictionaries, sets, arrays, & strings.

eg., 1) lst = [i for i in range(4)]
print(lst)
[0, 1, 2, 3]

2) lst = [i*i for i in range(4)]
print(lst)
[0, 1, 4, 9]

3) lst = [i*i for i in range(4) if i%2==0]
print(lst)
[0, 4]

4) lst = ["Milo", "Sarah", "Bruno", "Anny", "Rosa"]
onames = [i for i in lst if "o" in i]
print(onames)
['Milo', 'Bruno', 'Rosa'] → prints names with "o" in them

5) lst = ["Milo", "Sarah", "Bruno", "Anny", "Rosa"]
bignames = [i for i in lst if len(i) > 4]
print(bignames)
['Sarah', 'Bruno'] → prints names with >4 letters

• Imp Note | Set comprehension

Set comprehension is the same as list comprehension. This time we have sets instead of lists. Examples 1–5 mentioned above work for sets also.

- **List operations:**

e.g., list1 = ["A", "B"]
list2 = [10, 20, 30]

- **multiplication operator (*):**
  - print (list1 * 2) → ['A', 'B', 'A', 'B']
  - print (list2 * 2) → [10, 20, 30, 10, 20, 30]
- **Concatenation operator (+):**
  - list1 += "C"
  - print (list1) → ['A', 'B', 'C']
  - print (list1 + list2) → ['A', 'B', 10, 20, 30]

list1 += ["C"]

- **Length operator (len):**
  - print (len (list1)) → 2
  - print (len (list2)) → 3
- **min / max functions:**
  - print (min (list1)) → A
  - print (max (list1)) → B
  - print (min (list2)) → 10
  - print (max (list2)) → 30
- **sum function:**
  - print (sum (list1)) → TypeError: unsupported operand type(s) for +: 'int' and 'str'
  - print (sum (list2)) → 60
- **in operator:**
  - print ("A" in list1) → True
  - print ("D" in list1) → False
  - for i in list2:
    print i → 10
    20
    30

(90) a) Given 6 nos. [10, 20, 30, 40, 50, 60]. Find the numbers > 30 & count how many such nos. are there.

```
List 1 = [10, 20, 30, 40, 50, 60]
count = 0
for i in list 1:
    if i > 30:
        print ("Nos. > 30 :", i)
        count += 1
print ("No. of nos. > 30:", count)
```

```
Nos. > 30 : 40
Nos. > 30 : 50
Nos. > 30 : 60
No. of nos. > 30 : 3
```

- Function list:
- sort (): This method sorts the list in ascending order.

e.g., 1)
```
colors = ["Violet", "Blue", "Indigo", "Green"]
num = [2, 3, 5, 1, 4, 9, 8, 7]
colors.sort()
print (colors) → ["Blue", "Green", "Indigo", "Violet"]
num.sort()
print (num) → [1, 2, 3, 4, 5, 7, 8, 9]
```

- reverse: This method prints the reversed list. ~~in descending order~~

e.g., 1)
```
colors = ["Vio", "Ind", "Blu", "Gre"]
num = [4, 2, 5, 3, 6, 1, 2, 1, 2, 8, 9, 7]
colors.reverse() → ["Gre", "Blu", "Ind", "Vio"]
print (colors)
num.reverse()
print (num) → [7, 9, 8, 2, 1, 2, 1, 6, 3, 5, 2, 4]
```

- Sort & reverse: Using them simultaneously will first sort the list in ascending order & then reverse it, i.e., it shows the list in descending order.

e.g., 1) colors = ["Vio", "Red", "Blu", "Gre"]
num = [4, 2, 5, 3, 6, 1, 2, 1, 2, 8, 9, 7]
Colors.sort(reverse = True)
print(colors) → ['Vio', 'Red', 'Gre', 'Blu']
num.sort(reverse = True)
print(num) → [9, 8, 7, 6, 5, 4, 3, 2, 2, 2, 1, 1]

• append(): This method inserts objects passed to it at the end of the list.

e.g., 1) colors = ["Vio", "Blu", "Red"]
colors.append("Gre")
print(colors) → ['Vio', 'Blu', 'Red', 'Gre']

2) num = [1, 2, 3]
num.append(4)
print(num) → [1, 2, 3, 4]

• insert(): This method inserts an item at the given index. The user has to specify the index & the item to be inserted within insert() method.

e.g., 1) colors = ["Red", "Green", "Pink"]
colors.insert(2, "Blue")
print(colors) → ['Red', 'Green', 'Blue', 'Pink']

2) num = [5, 4, 3, 4, 5, 6, 4, 3]
num.insert(3, 10)
print(num) → [5, 4, 3, 10, 4, 5, 6, 4, 3]

• extend(): This method adds an entire list or any other collection of datatype (set, tuple, dictionary) to the existing list.

e.g., 1) num = [1, 2, 3]
num.extend([4, 5])
print(num) → [1, 2, 3, 4, 5]

2) num1 = [1, 2, 3]
num2 = [4, 5, 6]
num1.extend(num2)
print(num1) → [1, 2, 3, 4, 5, 6]

3) num = [1, 2, 3]
num.extend ((4,5,6))
print (num) → [1, 2, 3, 4, 5, 6]

4) num = [1, 2, 3]
num.extend (["four", "five", "six"])
print (num) → [1, 2, 3, 'four', 'five', 'six']

5) num = [1, 2, 3]
num.extend ("four")
print (num) → [1, 2, 3, 'f', 'o', 'u', 'r']

• count (): Returns the count of the number of items with the given value.

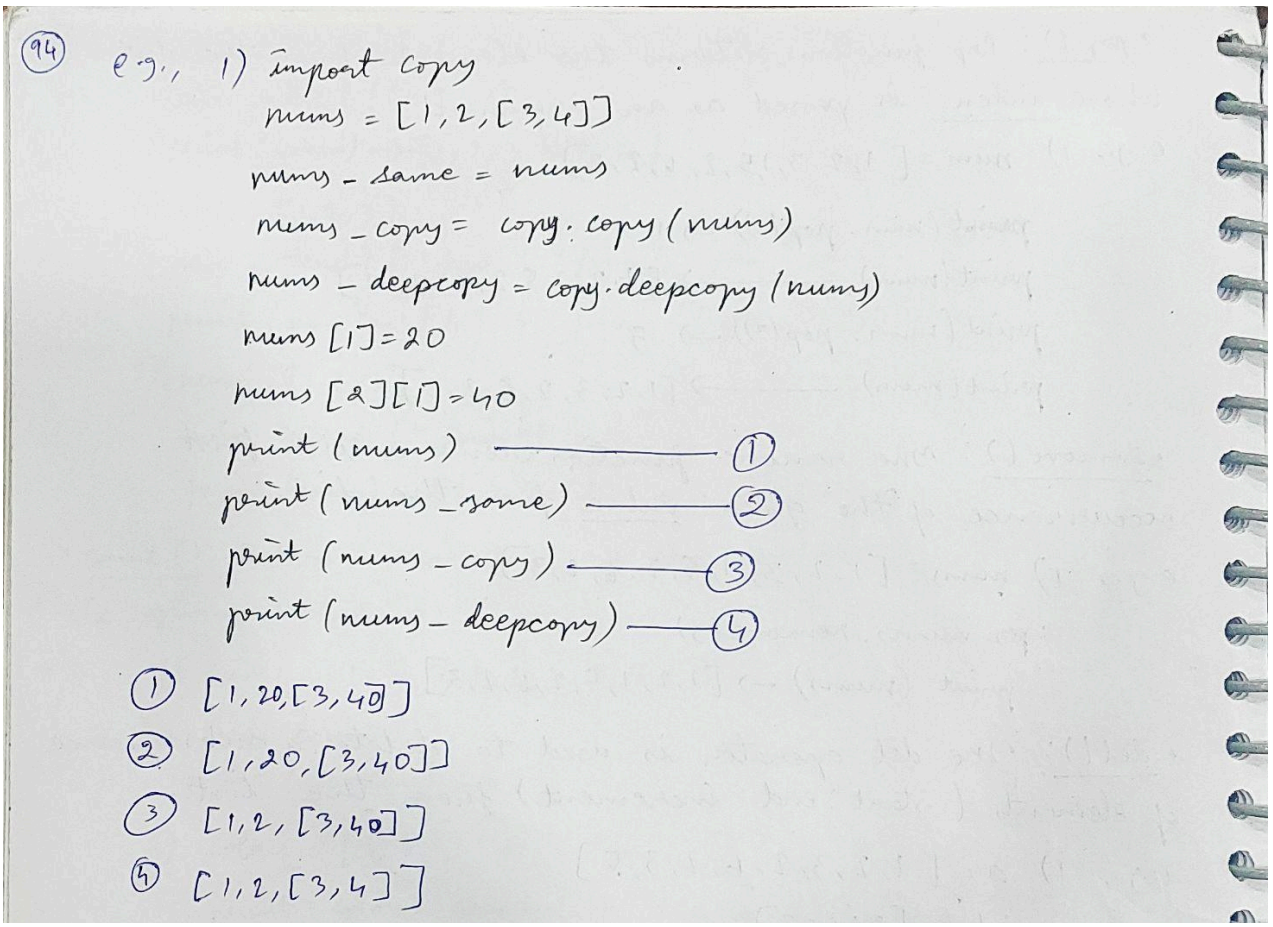
e.g., 1) num = [1, 2, 3, 1, 2, 4, 1, 5, 6]
counting = num.count(1)
print (counting) → 3

2) colors = ["V", "G", "I", "B", "G"]
print (colors.count("G")) → 2

• copy (): Returns a copy of the list. This can be done to perform operations on the list without modifying the original list.

e.g., 1) A program without copy:
colors = ["V", "G", "I", "B"]
newcolors = colors → They refer to the same list objects.
newcolors.append ("P")
print (newcolors) → ['V', 'G', 'I', 'B', 'P']
print (colors) → ['V', 'G', 'I', 'B', 'P']
} Any changes made to the list relates to both the names newcolors & colors.

What if we want to insert 'P' in the list without modifying the original list?
colors = ["V", "G", "I", "B"]
newcolors = colors.copy () → Creates a copy of colors
newcolors.append ("P")
print (newcolors) → ['V', 'G', 'I', 'B', 'P']
print (colors) → ['V', 'G', 'I', 'B']



import copy num1=[1,2,[3,4]] num1_same=num1 num1_copy=copy.copy(num1) num1_deepcopy=copy.deepcopy(num1) num1[1]=20 num1[2][1]=40 print(num1) print(num1_same) print(num1_copy) print(num1_deepcopy)	[1, 20, [3, 40]] [1, 20, [3, 40]] [1, 2, [3, 40]] [1, 2, [3, 4]]

What's happening?

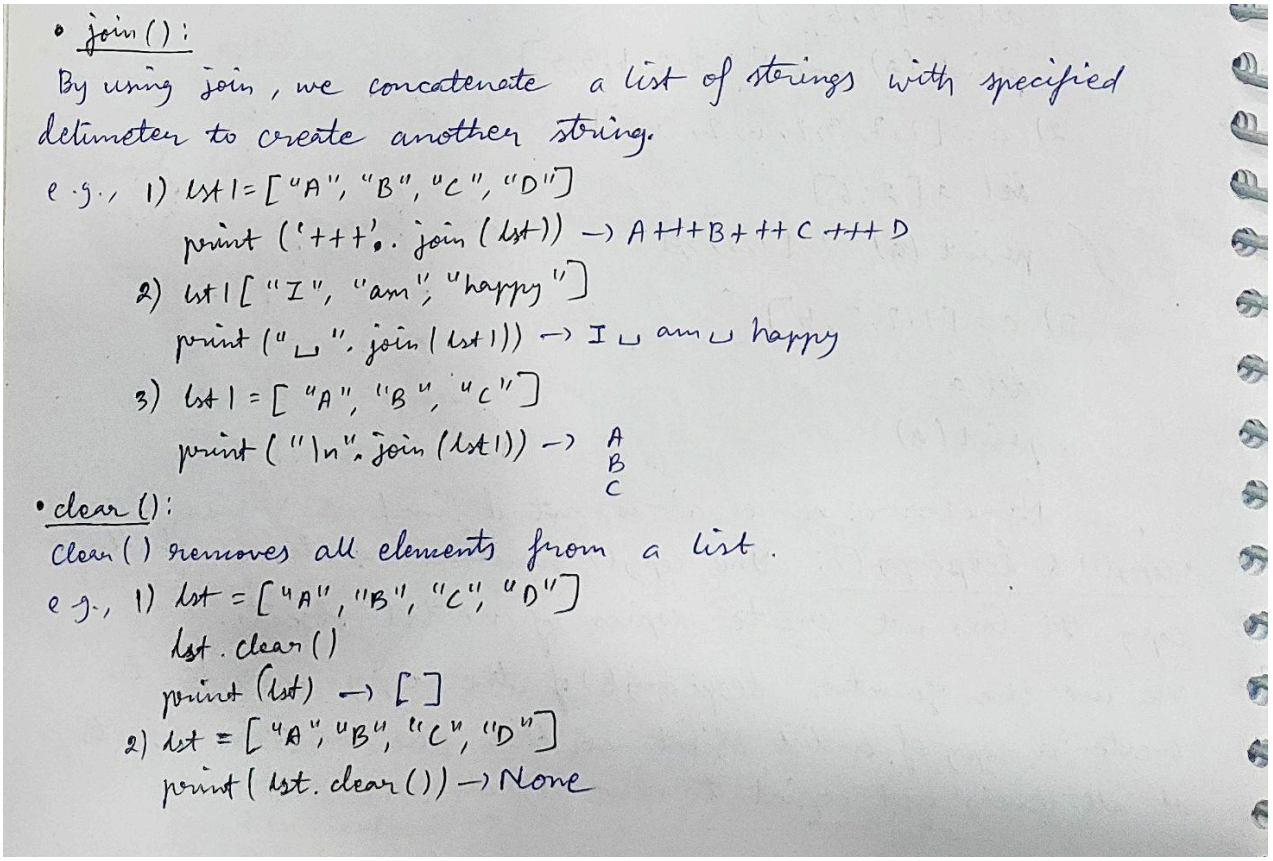
1) **print**(**num1**) - The print statement literally prints the updated **num1**.

2) **print**(**num1_same**) - **num1_same** is the same as **num1**. So the print statement literally prints the updated **num1**.

3) **print**(**num1_copy**) - **num1_copy** makes a shallow copy of **num1**. So 1 and 2 are a part of **num1_copy** and they remain fixed. By that I mean, it doesn't matter what change is happening in **num1**, 1 and 2 of **num1_copy** remain unchanged. However, since **num1_copy** makes a shallow

copy of **num1**, [3, 4] are not copied properly. So if any changes are happening in [3, 4] of **num1**, they are also reflected in **num1_copy**.

4) **print**(**num1_deepcopy**) – Here, the **entire num1** is deeply copied. So it doesn't matter what change is happening in **num1**, **num1_deepcopy** remains unaltered.



Lambda Function

• Lambda function:

In Python, a lambda function is a small anonymous function without a name. It is defined using the lambda keyword and has the following syntax:

lambda arguments : expression

Lambda functions are often used in situations where a small function is required for a short period of time. They are commonly used as arguments to higher-order functions, such as map, reduce, and filter.

e.g., 1) Write a program to find the square of a number.

i) Normal way:

```
def square(x):
    return x**3
print(square(5)) ⟶ 125
```

ii) Using lambda function:

```
square = lambda x : x**3
print(square(5)) ⟶ 125
```

Instead of defining a function & giving indentation & all that, the entire thing can be summed up in one line using lambda.

Lambda functions can have multiple arguments, just like regular functions.

e.g., 1)
```
avrg = lambda x, y, z : (x+y+z)/3
print(avrg(1, 2, 3)) ⟶ 2.0
```

Note: Functions can be passed as arguments of other functions.

e.g., 1)
```
def fn(x):
    return x**3
def appl(fn, value):
    return 6 + fn(value)
print(appl(fn, 2)) ⟶ 14
```

(98) Now, instead of defining fn as a function, we can use lambda as follows:

```
1) cube = lambda x: x**3
   def appl (fn, value):
       return 6 + fn(value)
   print (appl (cube, 2)) ⟶ 14
```

We don't even need to define cube.

```
2) def appl (fn, value):
       return 6 + fn (value)
   print (appl (lambda x: x**3, 2)) ⟶ 14
```

Here lambda is working as an anonymous function. We did not define cube, but it is acting anonymously like cube.

Map

(97)

• **Map, Filter, and Reduce:**

In Python, the map, filter, and reduce functions are built-in functions that allow us to apply a function to a sequence of elements and return a new sequence.
These functions are known as higher-order functions as they take other functions as arguments.

• **Map ():**

The map function applies a function to each element in a sequence & returns a new sequence containing the transformed elements.

Syntax: map (function, iterable)

The function argument is a function that is applied to each element in the iterable argument.
The iterable argument can be a list, tuple, or any other iterable object.

e.g., 1) Write a program to find the cubes of [1,2,3,4].

i) Without using map:

```
def cube (x):
    return x*x*x
nos = [1,2,3,4]
cubenos = []
for i in nos:
    cubenos.append (cube (i))
print (cubenos) ⟶ [1,8,27,64]
```

ii) Using map:

```
def cube (x):
    return x*x*x
nos = [1,2,3,4]
cubenos = list (map (cube, nos))
print (cubenos) ⟶ [1,8,27,64]
```

• Using lambda function:

```
nos = [1,2,3,4]
cubenos = list (map (lambda i:
                     i*i*3, nos))
print (cubenos) ⟶ [1,8,27,64]
```

Note: The output of map is to be brought in the form of a list. So we use list (map (cube, nos)).
If 'list' is not used here, output: <map object at 0x7f70d015f6a0>

Filter

(98)

- **Filter ():**

The filter function filters a sequence of elements based on a given predicate (a function that returns a boolean value) and returns a new sequence containing only the elements that meet the predicate.

Syntax: filter(predicate, iterable)

The predicate argument is a function that returns a boolean value and is applied to each element in the iterable argument.

e.g., 1) Print all numbers in [1,2,3,4,5,6,7] which are >4.

i) Without using filter:

```
nos = [1,2,3,4,5,6,7]
finalnos = []
for i in nos:
    if i>4:
        finalnos.append(i)
print(finalnos) ⟶ [5,6,7]
```

ii) Using filter:

```
nos = [1,2,3,4,5,6,7]
def noscondition(i):
    if i>4:
        return i
```

} instead of these 2 lines, we can also write `return i>4`

```
finalnos = list(filter(noscondition, nos))
print(finalnos) ⟶ [5,6,7]
```

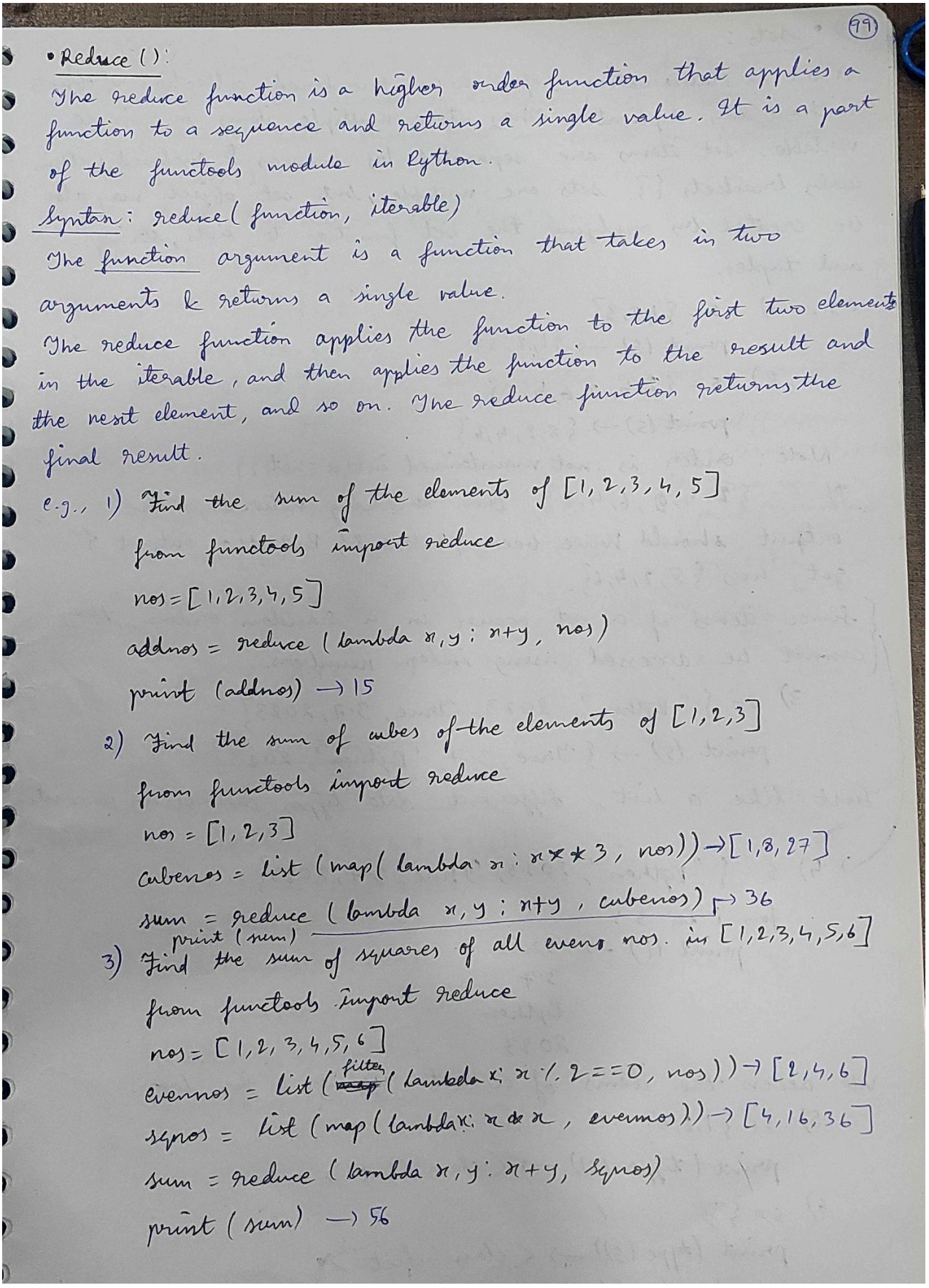
Note: If we don't use list, then the output is

<filter object at 0x7f1cff365370>

- **Using lambda function:**

```
nos = [1,2,3,4,5,6,7]
finalnos = list(filter(lambda i: i>4, nos))
print(finalnos) → [5,6,7]
```

Reduce



99

• Reduce ():

The reduce function is a higher order function that applies a function to a sequence and returns a single value. It is a part of the functools module in Python.

Syntax: reduce(function, iterable)

The function argument is a function that takes in two arguments & returns a single value.

The reduce function applies the function to the first two elements in the iterable, and then applies the function to the result and the next element, and so on. The reduce function returns the final result.

e.g., 1) Find the sum of the elements of [1, 2, 3, 4, 5]

```
from functools import reduce
nos = [1,2,3,4,5]
addnos = reduce ( lambda x,y : x+y, nos)
print (addnos) → 15
```

2) Find the sum of cubes of the elements of [1,2,3]

```
from functools import reduce
nos = [1,2,3]
cubenos = list (map( lambda x : x**3, nos)) → [1,8,27]
sum = reduce ( lambda x,y : x+y , cubenos) → 36
print (sum)
```

3) Find the sum of squares of all even nos. in [1,2,3,4,5,6]

```
from functools import reduce
nos = [1,2,3,4,5,6]
evennos = list (filter( lambda x: x %2==0, nos)) → [2,4,6]
sqnos = list (map (lambda x: x**x , evennos)) → [4,16,36]
sum = reduce ( lambda x,y: x+y, sqnos)
print ( sum) → 56
```

Set

(100) • Sets :

A set in Mathematics refers to an unordered collection of objects without any duplicates. They store multiple items in a single variable. Set items are separated by commas & enclosed within curly brackets {}. Sets are mutable, but set objects may also be created by applying the set function to lists, strings, and tuples.

e.g., 1) s = {1,2,3}
print (s) → {1,2,3}

2) s = {2,4,2,6,4,8}
print (s) → {8,2,4,6}

Note : order is not maintained in a set.

If s = {2,4,2,6,4,8}, then thinking naturally, the output should have been {2,4,6,8}. But the output I got was {8,2,4,6}.

Since items of a set occur in a random order, they cannot be accessed using index numbers.

3) s = {"Python", 2023, True, 3.7, 2023}
print (s) → {True, 3.7, 'Python', 2023}.

Just like a list, different data types can occur in a set.

4) s = {"Python", 2023, True, 3.7, 2023}
for i in s:
print (i) → True
3.7
Python
2023

We access the items of a set using a for loop.

5) s = {1,2,3}
print (type (s)) → <class 'set'>

6) s = {}
print (type (s)) → <class 'dict'>

The output should have been set, but we get dictionary.

This is because the syntax for a set & a dictionary are the same, and s = { } creates a dictionary, not a set.
The following code shows how we create an empty set:

7) s = set()
print(type(s)) → <class 'set'>

8) s = {1, 2, 3, 4}
print(s) → {1, 2, 3, 4}

9) s = set(1, 2, 3, 4)
print(s) → TypeError: set expected at most 1 argument, got 4

10) s = set((1, 2, 3, 4))
print(s) → {1, 2, 3, 4}

11) s = set([1, 2, 3, "A", "B", "C", 1, "C"])
print(s) → {1, 2, 3, 'B', 'C', 'A'}
for i in s:
print(i, end = " ") → 1 2 3 B C A

12) s = set(("Python", 2023, True, 3.7, 2023))
if True in s:
print("True in s")
else:
print("True not in s")
True in s

- Set methods & functions:
- min(), max(), sum(), len(), in:

e.g., 1) s = set((1, 3, 5, 4, 4, 9, 2, 3, 6))
print(s) → {1, 2, 3, 4, 5, 6, 9}
print(min(s)) → 1
print(max(s)) → 9
print(len(s)) → 7
print(3 in s) → True
print(8 in s) → False

(102) • add():

Used to add a single element to a set.

e.g., 1) vehicles = {"car", "bus", "bike", "plane"}
vehicles.add("scooter")
print(vehicles) → {'scooter', 'car', 'bike', 'plane', 'bus'}

2) c1 = {"A", "B", "C", "D"}
c1.add("A")
print(c1) → {"B", "C", "A", "D"} → {"A", "C", "B", "D"}
'A' is not added twice. Also, we don't get any error. This is because a set does not contain duplicates

• update():

It is used to add new elements in a set. The input argument of the update function can be an object such as list, set, range, or tuple. It adds items to an existing set.

e.g., 1) vehicles = {"car", "bus", "bike", "plane"}
vehicles.update(("scooter", "cycle", "truck"))
vehicles.update(["scooter", "cycle", "truck"])
vehicles.update({"scooter", "cycle", "truck"})
print(vehicles)

) All of them give the same output:

{'scooter', 'car', 'truck', 'bike', 'plane', 'cycle', 'bus'}

• update method can also add items into an existing set from a new set.

e.g., 2) vehicles1 = {"car", "bus", "bike", "plane"}
vehicles2 = {"bus", "scooter", "plane", "cycle"}
vehicles1.update(vehicles2)
print(vehicles1) → {'scooter', 'car', 'bike', 'plane', 'cycle', 'bus'}

• remove():

It is used to remove an element from a set.

e.g., 1) vehicles = {"car", "bus", "bike", "plane"}
vehicles.remove("bus")
print(vehicles) → {'bike', 'car', 'plane'}

2) S = {"car", "bus", "bike", "plane"}
S.remove("scooter")
print(S) → KeyError: 'scooter'

Thus, if we use remove() to delete an item not in the set, it raises an error.

• **discard():**

It is used to remove an element from a set.

e.g., 1) S = {"car", "bus", "bike", "plane"}
S.discard("bus")
print(S) → {'bike', 'car', 'plane'}

2) S = {"car", "bus", "bike", "plane"}
S.discard("scooter")
print(S) → {'bike', 'bus', 'car', 'plane'}

Thus, if we use ~~deluse~~ discard() to delete an item not in the set, then no error is raised.

• **pop():**

It is used to remove the last item of the set. However, we don't know which item gets popped as sets are unordered. But the popped item can be accessed if the pop() method is assigned to a variable.

e.g., 1) S = {"car", "bus", "bike", "plane"}
item = S.pop()
print(S) → {'bus', 'car', 'plane'}
print(item) → bike

• **del():**

del is not a method. It is a keyword which deletes the set entirely.

eg., 1) S = {"car", "bus", "bike", "plane"}
del S
print(S) → NameError: name 's' is not defined

• **clear():**

This method clears all items in the set & returns an empty set.

e.g., 1) S = {"car", "bus", "bike", "plane"}
S.clear()
print(S) → set()

(104) • copy():

The copy() function is used to create a separate copy of a set so that the changes in one copy are not reflected in the other. The working principle is the same as in the case of list.

e.g., 1) d1 = {0, 1, 2, 3, 7, 8, 9}
d2 = d1.copy()
d1.update({4, 5, 6})
print(d1) → {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(d2) → {0, 1, 2, 3, 7, 8, 9}

• union():

The union() method prints all items present in two sets. It returns a new set.

e.g., 1) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "heli", "bus"}
v3 = v1.union(v2)
print(v3) → {'scooter', 'bike', 'car', 'plane', 'heli', 'bus'}

• intersection():

The intersection() method prints only those items which are similar to both sets. It returns a new set.

e.g., 1) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "heli", "bus"}
v3 = v1.intersection(v2)
print(v3) → {'bus'}

2) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "heli"}
v3 = v1.intersection(v2)
print(v3) → set()

- **intersection_update():**

The intersection_update() method prints only those items which are similar to both sets. It updates into the existing set from another set.

e.g., 1) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "heli", "bus"}
v1.intersection_update(v2)
print(v1) → {'bus'}

2) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "heli"}
v1.intersection_update(v2)
print(v1) → set()

- **difference():**

The difference() method prints only those items which are present in the original set and not in both sets. It returns a new set.

$A - B = A - (A \cap B)$

e.g., 1) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "bus", "heli"}
v3 = v1.difference(v2)
print(v3) → {'car', 'bike', 'plane'}

2) v1 = {"car", "bus"}
v2 = {"car", "bus"}
v3 = v1.difference(v2)
print(v3) → set()

3) v1 = {"car", "bus"}
v2 = {"bike", "heli"}
v3 = v1.difference(v2)
print(v3) → {'bus', 'car'}

• difference_update():

The difference_update() method prints only those items which are present in the original set & not in both sets. It updates into the existing set from another set.

e.g., 1) V1 = {"car", "bus", "bike", "plane"}
V2 = {"scooter", "bus", "heli"}
V1.difference_update(V2)
print(V1) → {'car', 'plane', 'bike'}

2) V1 = {"car", "bus"}
V2 = {"car", "bus"}
V3.difference_update(V2)
print(V1) → set()

3) V1 = {"car", "bus"}
V2 = {"bike", "plane"}
V1.difference_update(V2)
print(V1) → {'car', 'bus'}

• symmetric_difference():

The symmetric_difference() method prints only items that are not similar to both sets. It returns a new set.

$A \triangle B = (A - B) \cup (B - A)$
$= (A \cup B) - (A \cap B)$

Let A = {1, 2, 3, 4} & B = {3, 4, 5, 6}.
Then, A - B = {1, 2} ; B - A = {5, 6} ; A △ B = {1, 2, 5, 6}

e.g., 1) V1 = {"car", "bus", "bike", "plane"}
V2 = {"scooter", "bus", "heli"}
V3 = V1.symmetric_difference(V2)
print(V3) → {'scooter', 'car', 'heli', 'bike', 'plane'}

2) V1 = {"car", "bus"}
V2 = {"car", "bus"}
V3 = V1.symmetric_difference(V2)
print(V3) → set()

3) v1 = {"car", "bus"}
v2 = {"bike", "plane"}
v3 = v1.symmetric_difference(v2)
print(v3) → {'car', 'bus', 'bike', 'plane'}

- **symmetric_difference_update():**

The symmetric_difference_update() method prints only items that are not similar to both sets. It updates into the existing set from another set.

e.g., 1) v1 = {"car", "bus", "bike", "plane"}
v2 = {"scooter", "bus", "heli"}
v1.symmetric_difference_update(v2)
print(v1) → {'scooter', 'car', 'heli', 'bike', 'plane'}

2) v1 = {"car", "bus"}
v2 = {"car", "bus"}
v1.symmetric_difference_update(v2)
print(v1) → set()

3) v1 = {"car", "bus"}
v2 = {"bike", "plane"}
v1.symmetric_difference_update(v2)
print(v1) → {'bike', 'bus', 'car', 'plane'}

- **subset & superset:**

For checking subset we use <=.
For checking superset we use >=.

e.g., 1) mul2 = {2,4,6,8,10,12}
mul4 = {4,8,12}
print(mul2 <= mul4) → False
print(mul2 >= mul4) → True
print(mul4.issubset(mul2)) → True
print(mul4.issuperset(mul2)) → False

- **set comprehension:**
set comprehension is the same as list comprehension. see pg (88) for more details.

Tuple

(108) • Tuples ( ):

- They are non-scalar data types.
- Like lists, they are ordered collection of data items. They store multiple items in a single variable.
- Unlike lists, they are immutable.
- Tuple items are separated by commas & enclosed by round brackets ( ).

e.g., 1) t1 = (1, 2, 3, 4)
t2 = ("R", "G", "B")
print (t1) → (1, 2, 3, 4)
print (t2) → ('R', 'G', 'B')

2) t1 = (1, 2, [3, 4], "abc", {5, 6})
print (t1) → (1, 2, [3, 4], 'abc', {5, 6})
print (type (t1)) → <class 'tuple'>

3) i) t1 = (1,) ← single value needs trailing comma → ii) t1 = (1)
print (t1) → (1,) print (t1) → 1
print (type (t1)) → <class 'tuple'> print (type (t1)) → <class 'int'>

4) t1 = (1, 2, 3)
t1[1] = 20
print (t1) → TypeError: 'tuple' object does not support item assignment

• Tuple operations: t1 = ("Mon", "Tue") t2 = (10, 20, 30)

• Multiplication (*): >>> t1 * 2
('Mon', 'Tue', 'Mon', 'Tue')

• Concatenation (+): >>> t3 = t1 + "Wed"
>>> t3
('Mon', 'Tue', 'Wed')

>>> t3 = t1 + t2
>>> t3
('Mon', 'Tue', 10, 20, 30)

• Length: >>> len (t1)
2

• Indexing: >>> t1[1]
'Tue'

>>> t2[-1]
30 [∵ t2[-1] = t2[len-1] = t2[3-1] = t2[2] = 30]

• Index range (start : end : increment) : >>> t2[0:2]
(10, 20)

>>> t1[1:2]
('T',)

>>> t2[0:-1]
(10, 20)

>>> t2[0::2]
(10, 30)

>>> t2[1:]
(20, 30)

- min/max: >>> min(t2) → 10 >>> max(t2) → 30
- sum: >>> sum(t2) → 60
- in: >>> "Fra" in t1 → False
- Manipulating tuples:

Since a tuple is immutable, to alter it in any way, we must convert it to a list. They we perform whatever operations we want on the list & then convert it back to a tuple.

e.g., 1) c1 = ("Ind", "USA", "Fra", "Pak") [w/o converting to list]
c1.append("Rus")
print(c1) → AttributeError: 'tuple' object has no attribute append

2) c1 = ("Ind", "USA", "Fra", "Pak") [After converting to list]
c2 = list(c1)
c2.append("Rus")
c2.pop(1)
c3 = tuple(c2)
print(c3) → ('Ind', 'Fra', 'Pak', 'Rus')

- Tuple functions:
- Zip() function: Zip() function is used to produce a zip object (iterable object), whose ith element is a tuple containing the ith element from each iterable object passed as an argument to the zip function. Since zip creates a collection of tuples, we apply set/list/tuple function to convert the zip object.

e.g., 1) c1 = ("Ind", "Pak", "Fra", "USA")
c2 = (10, 20, 30)
c3 = ('Win', 1, 'Lose', 2)
z1 = tuple(zip(c1, c2))
print(z1) → (('Ind', 10), ('Pak', 20), ('Fra', 30))
z2 = list(zip(z1, c3))
print(z2) → [(('Ind', 10), 'Win'), (('Pak', 20), 1), (('Fra', 30), 'Lose')]
z3 = set(zip(c1, c2, c3))
print(z3) → {('Fra', 30, 'Lose'), ('Ind', 10, 'Win'), ('Pak', 20, 1)}

(110) • count() :

The count method returns the number of times the given element appears in the tuple.

e.g., 1) t1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)

res = t1.count(2)

print(res) → 3

• index(): syntax to find index in a particular range: tuple.index(element, start, end)

The index method returns the index of the first occurrence of the given element in the tuple.

e.g. 1) t1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)

• res = t1.index(2)

print(res) → 2

• res = t1.index(5)

print(res) → ValueError : tuple.index(x) : x not in tuple

• res = t1.index(3, 4, 8) → we are going to find the index of 3 between the indices 4 to (8-1) = 7.

print(res) → 5

t1 : 0, 1, 2, 3, 2, 3, 1, 3, 2

index : 0, 1, 2, 3, 4, 5, 6, 7, 8

the first occurrence of

Dictionary

(111)

- **Dictionary :**
- A dictionary is an unordered sequence of key-value pairs. However, dictionaries are ordered from Python 3.7 onwards.
- They store multiple items in a single variable.
- The key-value pairs are separated by commas & enclosed within curly brackets { }.
- Relation with a literal dictionary : When we don't know the meaning of a word, we find it in a dictionary. The word itself will be the key & its meaning will be the value.
- Dictionary examples : ① storing the marks of a class of students.
② storing grades of students by their roll nos.
③ Mapping employee id with employee name, etc.

e.g. 1) `info = {"name": "Python", "ver": 3.0, "latest": True}`

`print(info)`

`{'name': 'Python', 'ver': 3.0, 'latest': True}`

`print(info["name"])` → Python

`print(info["latest"])` → True

accessing values → `print(info.values())` → `dict_values(['Python', 3.0, True])`

accessing keys → `print(info.keys())` → `dict_keys(['name', 'ver', 'latest'])`

accessing key-value pairs → `print(info.items())` → `dict_items([('name', 'Python'), ('ver', 3.0), ('latest', True)])`

2) Example of creating a dictionary from scratch.

```
details = {}
details[1] = "Jan"
details[2] = "Feb"
details[3] = "Mar"
details[4] = "Apr"
print(details) → {1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr'}
print(type(details)) → <class 'dict'>
print(details[4]) → Apr
print(details[5]) → KeyError : 5
```

for item in ~~data~~ details.keys(): → for item in 1,2,3,4
print(details[item]) → prints details[i], i=1,2,3,4

Jan
Feb
Mar
Apr

- Dictionary operations:

digits = {0: 'zero', 1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}

- length (len): >>> len(digits)
  6
- indexing: >>> digits[4]
  'four'

Q) cl = [{'name': 'A', 'age': 20}, {'name': 'B', 'age': 30}]
print(cl[0]['name']) → A
print(cl[1]['age']) → 30

- min/max functions: >>> min(digits) >>> max(digits)
  0 5
- sum: This function works assuming keys are compatible for addition.

>>> sum(digits)
15

- in: >>> 5 in digits >>> 10 in digits
  True False
- Note: in, min, max, and sum apply only to the keys in a dictionary.

- Dictionary methods:

- update():

This method updates the value of the key provided to it if the item already exists in the dictionary. Otherwise, it creates a new key-value pair.

e.g., 1) months = {1: 'Jan', 2: 'Feb', 3: 'Mar'}
print(months) → {1: 'Jan', 2: 'Feb', 3: 'Mar'}
months.update({2: 'Dec'})
print(months) → {1: 'Jan', 2: 'Dec', 3: 'mar'}

months.update({4:"Apr"})
print(months) → {1:'Jan', 2:'Dec', 3:'Mar', 4:'Apr'}

mon1 = {5:'May'}
months.update(mon1)
print(months) → {1:'Jan', 2:'Dec', 3:'Mar', 4:'Apr', 5:'May'}

• clear():

This method removes all key value pairs.

e.g., 1) nos = {1:'one', 2:'two'}
nos.clear()
print(nos) → {}

• pop():

This method removes a key-value pair whose key is passed as a parameter.

e.g., 1) nos = {'one':1, 'two':2}
nos.pop('one')
print(nos) → {'two':2}

2) nos = {'one':1, 'two':2}
nos.pop(2)
print(nos) → KeyError:2

• popitem():

This method removes the last key-value pair from the dictionary.

e.g., 1) nos = {'one':1, 'two':2, 'three':3}
nos.popitem()
print(nos) → {'one':1, 'two':2}

• del():

This method removes any particular dictionary item.

e.g., 1) nos = {'one':1, 'two':2, 'three':3}
del nos['two']
print(nos) → {'one':1, 'three':3}

If the key is not provided, then the del keyword will delete the dictionary entirely.

e.g., 1) nos = {'one':1, 'two':2}
del nos
print(nos) → NameError: name 'nos' is not defined

(114) • Inverted Dictionary :

Given any dictionary (D), an inverted dictionary (D^{-1}) is a dictionary s.t. the keys of D^{-1} are the values of D & the values of D^{-1} are the keys of D.

So, if D = {'dubious': 'doubtful', 'hilarious': 'amusing'},
then D^{-1} = {'doubtful': ['dubious'], 'amusing': ['hilarious']}

• What happens if different keys have same values?

(2) If D = {'dubious': 'doubtful', 'hilarious': 'amusing',
'suspicious': 'doubtful', 'comical': 'amusing'}
then D^{-1} = {'doubtful': ['dubious', 'suspicious'], 'amusing': ['hilarious', 'comical']}.

Code :

```
1) my_dict = {'dubious': 'doubtful', 'hilarious': 'amusing',
              'suspicious': 'doubtful', 'comical': 'amusing'}
2) invd = dict()
3) for key, value in my_dict.items():
4)     invd.setdefault(value, list()).append(key)
5) print(invd)
```

1) Creating a dictionary called my_dict.
2) We create an empty inverted dictionary & name it invd.
3) We are looping over all keys & values in ~~the~~ my_dict.
4) setdefault allows us to set some keys with default values. The keys of invd are the values of my_dict. The values of invd will be returned as a list (as there can be multiple values for a single key). The values of invd will be the keys of my_dict & they will be appended in invd (which was originally empty) one by one.
5) invd is {'doubtful': ['dubious', 'suspicious'], 'amusing': ['hilarious', 'comical']}

Some Searching and Sorting Algorithms in Python

◍ Linear Search

Here, we search for a chosen element in a linear way (compare it one by one to each and every element in the list).

l=[6,9,5,4,8,2,3]
key=10
def lins(l,key):
  for i in range(len(l)):
  if key==l[i]:
  return f"{key} found at index {i}"
  return f"{key} not found"
print(lins(l,key))
	10 not found

l=[6,9,5,4,8,2,3]
key=8
def lins(l,key):
  for i in range(len(l)):
  if key==l[i]:
  return f"{key} found at index {i}"
  return f"{key} not found"
print(lins(l,key))
	8 found at index 4

◍ Binary Search

Here, we search for an element (x) in a given list. This is a recursive algorithm. Steps:

1) Sort the list.
2) In the present list, set low=0 (least index) and high=len(l)-1 (highest index).
3) Observe the mid index. If x=l[mid], then we have found the location of x. If it is not the case, then obviously either x<l[mid] or .x>l[mid].
4) If x<l[mid], we recursively call the algorithm by updating the value of high as mid - 1.
5) If x>l[mid], we recursively call the algorithm by updating the value of low as mid + 1.

def bsearch(l, low, high, x):
  if high >= low:
  mid = (high + low) // 2
  if x==l[mid]:
  return mid
  elif x<l[mid]:
  return bsearch(l, low, mid - 1, x)
  else:
  return bsearch(l, mid + 1, high, x)
  else:
  return -1
l=[9,4,8,2,6,7]
l.sort()
x = 8
result = bsearch(l, 0, len(l)-1, x)
if result!=-1:
  print(f"{x} is present at index {str(result)}")
else:
  print(f"{x} is not present in the list")
	8 is present at index 4

def bsearch(l, low, high, x):
  if high >= low:
  mid = (high + low) // 2
  if x==l[mid]:
  return mid
  elif x<l[mid]:
  return bsearch(l, low, mid - 1, x)
  else:
  return bsearch(l, mid + 1, high, x)
  else:
  return -1
l=[9,4,8,2,6,7]
l.sort()
x = 10
result = bsearch(l, 0, len(l)-1, x)
if result!=-1:
  print(f"{x} is present at index {str(result)}")
else:
  print(f"{x} is not present in the list")
	10 is not present in the list

◍ Selection Sort

Here, we set the index 0 element as the minimum element.
Comparing it with the other elements in the list, we keep finding the minimum element and put it at the beginning of the list until it is sorted.
Here, the elements are sorted in ascending order.

l=[4,5,3,2,1]
for i in range(0,len(l)):
  min1=i
  for j in range(i+1,len(l)):
  if l[j]<l[min1]:
  min1=j
  l[i],l[min1]=l[min1],l[i]
print(l)
	[1, 2, 3, 4, 5]

l=[4,5,3,2,1]
for i in range(0,len(l)):
  min1=i
  for j in range(i+1,len(l)):
  if l[j]<l[min1]:
  min1=j
  l[i],l[min1]=l[min1],l[i]
  print(l)
	[1, 5, 3, 2, 4]
	[1, 2, 3, 5, 4]
	[1, 2, 3, 5, 4]
	[1, 2, 3, 4, 5]
	[1, 2, 3, 4, 5]

◍ Bubble Sort

Here, we repeatedly swapping the adjacent elements until the list is sorted.
Here, the elements are sorted in descending order (the heaviest bubble settles down first in its correct place, followed by the second heaviest bubble, and so on until the lightest bubble settles in its correct place at the last step).

l=[5,4,2,3,1]
for i in range(len(l)-1):
  for j in range(len(l)-i-1):
  if l[j]>l[j+1]:
  l[j],l[j+1]=l[j+1],l[j]
print(l)
	[1, 2, 3, 4, 5]

l=[5,4,2,3,1]
for i in range(len(l)-1):
  for j in range(len(l)-i-1):
  if l[j]>l[j+1]:
  l[j],l[j+1]=l[j+1],l[j]
  print(l)
	[4, 2, 3, 1, 5]
	[2, 3, 1, 4, 5]
	[2, 1, 3, 4, 5]
	[1, 2, 3, 4, 5]

File Handling

(128) • File Handling:

• Whenever we are programming on IDLE or some online compiler, the data provided by us remain in memory only during the lifetime of the program.

• To store data permanently, we store them in files, which are streams of bytes comprising data of interest.

• Rather than opening a file, typing in it, and saving it (all manually), we can do the same things by simply typing the relevant codes on the compiler.

• open():

Before performing a read or write operation in a file, we need to open it first. The built-in function open() is used for this purpose.

Syntax: f = open(name, mode)

→ The file object returned by the function open is named 'f'. Thus, we will use this name to perform all operations.

The first argument is the name of the file.

The second argument is the mode for accessing the file.

There are 3 modes: read (r), write (w), append (a).

• Note

• Read() mode is used when an existing file is to be read. It is a default mode, i.e., if no mode is mentioned, Python by default chooses to read.

• Write() mode is used when a file is to be accessed for writing data in it.

• Append() mode is used to write into a file by appending contents at the end of the specified file.

• While opening a file in the read() mode, if the specified file does not exist, it would lead to an error. (You cannot read something which doesn't exist).

• While opening a file in the write() mode, if the specified file does not exist, Python would create a new file. (If you want to write something, and you don't have a paper right in front of you, you go get it somewhere else and start writing).

- While opening a file in the write() mode, if the specified file already exists, the file gets overwritten. (If you want to write something, but the piece of paper that you have already has something written on it, then you overwrite).
- While opening a file in the append() mode, if the specified file does not exist, a new file is created.
- The absence of the second argument in the open function sets it to default value 'r' (read mode).

e.g., 1) f = open ("Python.py", "w")

This creates a new Python file called "Python", in which I want to write something.
If we are using an online compiler, the python file will be available in the directory where we are working. However, it may not be readily available when we are working on our systems (say, using IDLE). To find out the specific directory, type the following:

```
>>> from inspect import getsourcefile
>>> from os.path import abspath
>>> abspath(getsourcefile(lambda:0))
'C:\\Users\\AppData\\...\\Python311\\<pyshell#12>'
```

f.write("This is the first line") → We have written this line in 'Python'.
We are not typing it. → This is the number of characters written into the file.
It is output.
Now we want to output what we have written. To read what we have written, we can use read(), but it will give an error here because the file is already opened in write mode.

```
print(f.read())
```

io.UnsupportedOperation: not readable

To read the file, we need to open it first.

```
f = open("Python.py", "r")
print(f.read()) → This is the first line.
```

f.close() → 'Python.py' is closed now.

We use the close operation when a file is no longer required for reading or writing. The function close also saves the file.

f.write("Trial line") → ValueError: I/O operation on closed file

Once a file is closed, it cannot be read or written any further unless it is opened again.

```
f = open("Python.py", "w") # We want to write some new things in Python.py
f.write("This is the second line. It replaces the first line.")
f = open("Python.py", "r")
print(f.read()) → This is the second line. It replaces the first line.
f.close()
f = open("Python.py", "a") # Now we want to write something in Python.py without deleting the existing lines.
f.write("\n This is the appended line. The previous lines remain unaltered.")
f = open("Python.py", "r")
print(f.read()) → This is
```

→ This is the second line. It replaces the first line.
This is the appended line. The previous lines remain unaltered.

```
f.close()
f = open("Python.py", "w")
f.write('''Python:
Python is an interactive language. It has simple syntax.''')
f = open("Python.py", "r")
print(f.read()) →
```

→ Python:
Python is an interactive language. It has simple syntax.

• tell

The read() function retrieves the contents of the entire file.
The tell() function allows us to read data in smaller chunks.
It returns the current position in a file, which is specified in bytes.

e.g. f = open("Python.py", "r")

f.tell() ⟶ 0. We use tell to know the current position while reading the file object f. Initially, the current position of the file object f is set at 0, i.e. position of the first byte.

print(f.read(5))

↓

Pytho

(The output is the first 5 bytes starting from point 0).

print(f.tell) → 5. The current position of the file object f is set at 5.

print(f.read(6)) → n:
Pyt

• Readline: The readline() function allows us to read one line at a time.

e.g. f = open("Python.py", "r")

print(f.readline()) → Python:

print(f.readline()) → Python is an interactive ~~Programming~~ language. It has simple syntax.

print(f.readline()) → <Blank space>. Note that when all the contents of a file have been read, a further read operation will return a null string.

• seek: We use seek() function to reach the desired position in a file, which is specified in bytes.

e.g. f = open("Python.py", "r")

f.seek(5) → current position is now at 5.

print(f.read(6)) → n:
Pyt

• Writeline: The writeline() function takes a list of lines to be written in the file as an argument.

```
f= open ("Python.py", "w")
desc = ['This is a new sentence.', 'We are using writelines.',
        'Let's see what happens.']
f.writelines(desc)
f = open ("Python.py", "r")
print (f.read()) → This is a new sentence. We are using
                   writelines. Let's see what happens.
```

• Copy: The copy() function is used to copy the contents of a text file to another text file.

e.g.
```
f1 = open ("Python.py", "r")
f2 = open ("PythonCopy.py", "w")
data = f1.read()
f2.write(data)
f1 = open ("Python.py", "r")
print (f1.read())
print ()
f2 = open ("Python.py", "r")
print (f2.read())
```

This is a new sentence. We are using writelines. Let's see what happens.

This is a new sentence. We are using writelines. Let's see what happens.

• Flush: The flush() function is used to save the contents of a file while we are still working on it.

Syntax: f.flush()

• Truncate: Used to truncate a file to a specific size. It should be used immediately after 'w' or 'a' operations w/o reading the file.

e.g.,
```
f = open("New1.py", "w")
d = ["This is a new sentence.", "We are using writelines,"]
f.writelines(d)
f.truncate(30)
f = open("New1.py", "r")
print (f.read()) → This is a new sentence. We are
```

Writing structures to a file:

Now, we want to write structures such as list, set, and dictionary in our python files without opening them.

- This can be done using the Pickle module.
- Pickling refers to the process of converting the structure to a byte stream before writing to the file.
- While reading the contents of the file, a reverse process called nonpickling is used to convert the byte stream back to the original structure.

e.g.,

```
import pickle
f = open("try1.py", "wb")
pickle.dump(['A', 1, True, 1.1, ['B', 'C']], f)
pickle.dump({'one':1, 2:'two', 'Python':True}, f)
pickle.dump({1,2,3,4,5}, f)
```

In Python 3.x versions, binary mode is used for reading or writing pickled data. So a file involving reading or writing pickled data should be opened in binary mode by specifying 'b' as the mode for opening it.

At this point, the list, dictionary, and set mentioned in steps 1, 2, and 3 are now inserted in try1.py

What if we want to print the contents of f? Let's try.

```
f = open("try1.py", "rb")
print(f.read()) -> b'\x80\x04\x95!\x00\x00\x00 ---
                    --- x04K\x05\x90.'
```

Python is giving the output in binary form which is unreadable.

So we try something different.

```
f = open("try1.py", "rb")
v1 = pickle.load(f)
v2 = pickle.load(f)
v3 = pickle.load(f)
print(v1, "\n", v2, "\n", v3) -> ['A', 1, True, 1.1, ['B', 'C']]
                                 {'one':1, 2:'two', 'Python':True}
                                 {1,2,3,4,5}
```

Errors and Exceptions

(34) • Errors & Exceptions :

• Errors (drumroll) occur when something goes wrong. Things can go wrong at two stages – during compilation & during execution.

• Errors that occur during compilation are called syntax errors.

• Errors that occur during execution are called Exceptions.

• Sometimes, a program may neither contain any syntax error, nor report any runtime error. However, we may still get wrong results. The errors responsible for creating wrong results are called logical errors or semantic errors.
We can track Logical errors using debugger.

We have already faced different types of errors in the past. Let us see some examples:

① NameError | This is an exception which occurs when a name (↓) appearing in the statement (or variable or function or module) does not exist, is not defined in the current scope or globally, or has not been assigned a value.

e.g. 1) print (sqrt (64))

NameError : name 'sqrt' is not defined

2) a = 10
print (b)

NameError : name 'b' is not defined

3) def study ():
print ("We are studying Python")

i) lol() ⟶ NameError : name 'lol' is not defined

ii) studi () ⟶ NameError : name 'studi' is not defined. Did you mean : 'study'?

4) n = Int(input ("Enter n :"))
print (n)

NameError : name 'Int' is not defined. Did you mean : 'int'?

• Note : The above error is not detected as a syntax error as we can define a function called Int as follows:

```
def int():
    return int(input("Enter n:"))
n = int()
print(n)
```

Enter n : 10
10

② TypeError | This is an exception which occurs when an operation is performed using an incompatible data type.

e.g., 1) n = 7
print("In a week there are " + n + "days")

TypeError: can only concatenate str (not "int") to str

2) lst = [1, 2, 3, "a"]
print(sum(lst))

TypeError: unsupported operand type(s) for +: 'int' and 'str'

3) lst = [1, 2, 3, "a"]
print(lst["a"])

TypeError: list indices must be integers or slices, not str.

③ ValueError | This is an exception which occurs when Python receives an inappropriate value for a correct data type.

e.g. 1) import math
n = -64
root = math.sqrt(n)
print(root)

ValueError: math domain error

2) str1 = "Hi"
x = int(str1)
print(x)

ValueError: invalid literal for int() with base 10: 'Hi'

④ **ZeroDivisionError** | This is an exception which occurs when division is performed with denominator = 0.

e.g., 1) `a,b = input("Enter a & b :").split()`
`print(int(a)/int(b))`

Enter a & b : 10 0

ZeroDivisionError : division by zero

⑤ **AttributeError** | This is an exception which occurs when an attribute reference or assignment fails. This means that the object has no attribute or method that is being called on it.

e.g. 1) `n = 123`
`print(n.capitalize())`

AttributeError : 'int' object has no attribute 'capitalize'

⑥ **ImportError** | This is an exception which occurs when we try to import a module that cannot be found or is not accessible in the current environment. It may be possible that the module is not installed, or the path has not been configured correctly, or the module has been misspelled, etc.

e.g., 1) `import pandas`

ModuleNotFoundError : No module named 'pandas'
(It has not been installed yet)

2) `import pamdas`

ModuleNotFoundError : No module named 'pamdas'
(It is installed, but misspelled)

⑦ **IOError / OSError** | This is an exception which occurs when an input or output operation fails. This can be caused due to system related errors such as insufficient disk storage, opening a non-existent file, trying to access a file with insufficient permissions, trying to access a file that is currently being used by other operations, etc.

e.g. 1) `f = open("file1.py", "r")`

FileNotFoundError : [Errno 2] No such file or directory : 'file1.py'

⑧ IndexError | This is an exception which occurs when we try to access an index that is out of a valid range.

e.g. 1) `n = ['a', 'b', 'c']`
`print(n[10])`
IndexError : list index out of range

2) `n = ['apple', 'ball', 'cat']`
`print(n[1][6])`
IndexError ; string index out of range

⑨ SyntaxError | This error occurs due to an invalid line of code.

e.g. 1) `print("Python)`
SyntaxError ; unterminated string literal

2) `age = 20`
`if age > 17`
`print("Age >= 18")`
SyntaxError ; expected :

3) `def sq(x:`
`return x * x`
`print(sq(4))`
SyntaxError : '(' was never closed

There are many other different types of errors (Exceptions).

38 • Handling Exceptions using Try... Except:

Python, being an interpreted language, terminates a program abruptly when it encounters an exception during execution. What if we have to print something written after the part where the exception exists, but we are not able to do so because Python stops at the execution stage?

e.g. 1)
```
def div(a,b):
    print(f"Quotient is {a/b}")
a = int(input("Enter a:"))
b = int(input("Enter b:"))
div(a,b)
print("I have to print this")
```

Output 1
```
Enter a: 10
Enter b: 5
Quotient is 2.0
I have to print this
```

Output 2
```
Enter a: 10
Enter b: 0
ZeroDivisionError: division by zero
```

To stop such things from happening, we use Try... Except clause.
The Try block comprises statements that have the potential to raise an exception.
The Except block describes the action to be taken when an exception is raised.

e.g. 1)
```
def div(a,b):
    print(f"Quotient is {a/b}")
a = int(input("Enter a"))
b = int(input("Enter b"))
try:
    div(a,b)
except ZeroDivisionError:
    print("Cannot divide by 0")
print("I have to print this")
```

```
Enter a: 10
Enter b: 0
Cannot divide by 0
I have to print this
```

The description of the exception can be printed as follows:

e.g. 1)
```
def div(a,b):
    print(f"Quotient is {a/b}")
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
  div(a,b)
except ZeroDivisionError as xyz:
  print("Cannot divide by 0\t", xyz)
print("I have to print this")
```

```
Enter a: 10
Enter b: 0
Cannot divide by 0        division by zero
I have to print this
```

It is also possible to track the exception raised by Python using the expression sys.exc_info(), which yields the details of the exception as a tuple comprising • the type of exception, • the description of the exception, and • a reference to the exception object.

e.g. 1)
```
import sys
def div(a,b):
    print(f"Quotient is {a,b}")
a = int(input("Enter a"))
b = int(input("Enter b"))
try:
    div(a,b)
except ZeroDivisionError as xyz:
    print("Cannot divide by 0\n", xyz)
    print(sys.exc_info())
print("I have to print this")
```

- Enter a: 10
- Enter b: 0
- Cannot divide by 0
- division by 0
- (<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero'), <traceback object at 0x000001233D5E7080>)
- I have to print this

(140) Sometimes, we may encounter problems having multiple types of errors (exceptions). Let us deal with one such problem:

e.g. 1)
```
import sys
def main():
    price = input("Enter price:")
    weight = input("Enter weight:")
    if price == '': price = None
    price = float(price)
    if weight == '': weight = None
    weight = float(weight)
    assert price >= 0 and weight >= 0
    result = price / weight
    print("Price per unit weight:", result)
    print("we are done")
if __name__ == '__main__':
    main()
```

Output 1

```
Enter price: 400
Enter weight: 0
```
ZeroDivisionError: float division by zero

Output 2

```
Enter price: 400
Enter weight: z
```
ValueError: could not convert string to float: 'z'

Output 3

```
Enter price: -400
Enter weight: 10
```
AssertionError

Output 4

```
Enter price: 400
Enter weight: <press Enter>
```
TypeError: float() argument must be a string or a real number, not 'NoneType'

To deal with all exceptions simultaneously (without any specific one in mind), we can place the targeted group of statements in the try block and specify an empty except clause. Such an except clause is capable of catching all possible errors.

eg. 1)
```
import sys
def main():
    price = input("Enter price:")
    weight = input("Enter weight:")
    try:
        if price == '': price = None
        price = float(price)
        if weight == '': weight = None
        weight = float(weight)
        assert price >= 0 and weight >= 0
        result = price/weight
        print("price per unit weight :", result)
    except:
        print("There is an error")
    print("We are done")
if __name__ == '__main__':
    main()
```

Outputs

Enter price : 400	400	-4100	<press Enter>
Enter weight : 0	3	10	10
We encountered an error			
We are done			

(142) Another way to deal with multiple exceptions is to enumerate all possible exceptions and include them in an except clause.

eg., 1)
```
import sys
def main():
    price = input("Enter price :")
    weight = input("Enter weight :")
    try:
        if price = "": price = None
        price = float(price)
        if weight = "": weight = None
        weight = float(weight)
        assert price >= 0 and weight >= 0
        result = price / weight
        print("Price per unit weight", result)
    except(ZeroDivisionError, ValueError, TypeError, AssertionError) as err:
        print("There is an error :\t", err)
        print(sys.exc_info())
    print("we are done")
if __name__ == '__main()__':
    main()
```

Output 1

- Enter price : 400
- Enter weight : 0
- There is an error : float division by zero
- (<class 'ZeroDivisionError'>, ZeroDivisionError('float division by zero'), <traceback object at 0x00000299ABD911E00>)
- We are done

Output 2

- Enter price : 400
- Enter weight : 3

- There is an error; could not convert string to float: 'z'
- (<class 'ValueError'>, ValueError("could not convert string to float: 'z'"), <traceback object at 0x00000231C716B380>)
- We are done

Output 3

- Enter price: <press Enter>
- Enter weight: 10
- There is an error; float() argument must be a string or a real number, not 'NoneType'
- (<class 'TypeError'>, TypeError("float() argument must be a string or a real number, not 'NoneType'"), <traceback object at 0x00000D435D47300>)
- We are done

Output 4

- Enter price: -400
- Enter weight: 10
- There is an error:
- (<class 'AssertionError'>, AssertionError(), <traceback object at 0x00000266DEB86A80>)
- We are done

Output 5

- Enter price: <press Enter>
- Enter weight: z
- TypeError...

Output 6

- Enter price: z
- Enter weight: 0
- ValueError

Output 7

- Price: -400
- Weight: <Enter>
- TypeError

Output 8

- Price: -10
- Weight: z
- ValueError

(144) Sometimes it is preferable to handle different exceptions separately. An empty except clause may be used to catch all other possible exceptions (e.g., invalid range).

e.g., 1)
```
import sys
def main():
    price = input("Enter price :")
    weight = input("Enter weight :")
    try:
        if price == '' : price = None
        price = float(price)
        if weight == '': weight = None
        weight = float(weight)
        assert price >= 0 and weight >= 0
        result = price / weight
    except ValueError:
        print("There is ValueError").
    except TypeError:
        print("There is TypeError")
    except ZeroDivisionError:
        print("There is ZeroDivisionError")
    except:
        print(str(sys.exc_info()))
    else:
        print("Price per unit weight :", result)
    print("We are done")
if __name__ == '__main__':
    main()
```

Output 1
- Price: 400
- Weight: 3
- There is ValueError
- We are done

Output 2
- Price: 400
- Weight: <press Enter>
- There is TypeError
- We are done

Output 3
- Price: 400
- Weight: 0
- There is ZeroDivisionError
- We are done

Output 4

- Price : -400
- Weight : 10
- (<class 'AssertionError'>, AssertionError (), <traceback object at 0x00000 1A F1 CF98B80>)
- We are done

Output 5

- Price : -400
- Weight : 0
- (<class 'AssertionError'>, AssertionError (), <traceback object at 0x00000 1A 25B48 6B80>)
- We are done

Note | In output 5, the line assert price >=0 --- and the line result = price / weight can raise exceptions. However, the assert statement is encountered first, and thus in the try ... except clause, AssertionError exception is raised. Since none of the mentioned exceptions match the AssertionError exception, the empty except block is executed.

Note | The segment of the script which handles an exception is called a handler. In the previous examples we saw that exceptions were raised & handled within a try ... except block. Remember the concept of scope - if a variable is not defined in a space, then python goes to the outer space to look for it. If it is not found in the immediate outer space, then python moves ahead further & so on. Try ... except works in the same way. If an exception is raised in the try block, but the script doesn't include the corresponding handler in the block, then python searches for the handler in the outer try ... except block & so on. PTO for example:

(146)
```
import sys
def main():
    price = input("Enter price:")
    weight = input("Enter weight:")
    try:
        if price == '': price = None
        try:
            price = float(price)
        except ValueError:
            print("There is a ValueError")
        if weight == '': weight = None
        try:
            weight = float(weight)
        except ValueError:
            print("There is a ValueError")
        assert price >= 0 and weight >= 0
        result = price / weight
        print("Price per unit weight:", result)
    except (ZeroDivisionError, TypeError, AssertionError) as err:
        print("There is an error:\t", err)
    except:
        print(sys.exc_info())
    print("We are done!")
if __name__ == "__main__":
    main()
```

Output 1

```
Enter price: m
Enter weight: 20
There is ValueError
There is an error: '>=' not supported between instances of
                   'str' and 'int'
We are done!
```

Output 2

Enter price : 20
Enter weight : 0
There is an error : float division by zero
We are done !

Output 3

Enter price : <press Enter>
Enter weight : 50
There is an error : float() argument must be a string or a real number, not 'NoneType'
We are done !

Note | In Output 1, there is a ValueError which Python handles in the inner try... except block.
However, when ZeroDivisionError & TypeError are raised in Output 2 and Output 3 respectively, Python does not find any matching except clause in the inner try ... except block. Thus, the search for the corresponding handler continues in the outer try ... except block.

48) • 'finally' keyword :

The finally code block is a part of exception handling. It is generally included after the try ... except block to execute some action irrespective of whether an exception is raised. Its output can be clearly understood if it is used inside a function rather than outside of it.

e.g. 1) using 'finally' outside a function :

i) Using 'finally'

```
try:
  n = int(input("Enter n: "))
except ValueError:
  print("Error; Enter an integer")
else:
  print(n," is an integer")
finally:
  print("Run this line always")
```

Output 1

Enter n: 20
20 is an integer.
Run this line always.

Output 2

Enter n: y
Error: Enter an integer
Run this line always

ii) Not using 'finally'

```
try:
  n = int(input("Enter n: "))
except ValueError:
  print("Error; Enter an integer")
else:
  print(n," is an integer")
print("Run this line always")
```

Output 1

Enter n: 20
20 is an integer
Run this line always

Output 2

Enter n: w
Error: Enter an integer
Run this line always

It can be seen from (i) & (iii) that their outputs are the same. So the role of 'finally' is not conclusive in this case. Let us try other examples, this time using functions.

2) Using 'finally' inside a function :

i) Using 'finally'

```
def f():
  try:
    n = int(input("Enter n:"))
    return 0
  except ValueError:
    print("There is an error")
    return 1
  finally:
    print("Run this line always")
print(f())
```

Output 1

```
Enter n: 20
Run this line always
0
```

Output 2

```
Enter n: m
There is an error
Run this line always
1
```

ii) Not using 'finally'

```
def f():
  try:
    n = int(input("Enter n:"))
    return 0
  except ValueError:
    print("There is an error")
    return 1
print("Run this line always")
print(f())
```

Output 1

```
Enter n: 20
0
```

Output 2

```
Enter n: m
There is an error
1
```

3)

```
def main():
  marks = int(input("Enter marks:"))
  try:
    assert marks>=0 and marks<=100
  except:
    print("Marks out of range")
  finally:
    print("Bye")
  print("Continue the program")
if __name__=="__main__":
  main()
```

Output 1

```
Enter marks: 200
Marks out of range
Bye
Continue the program
```

Output 2

```
Enter marks: 50
Bye
Continue the program
```