A Short Note on

Python Programming

September 2024

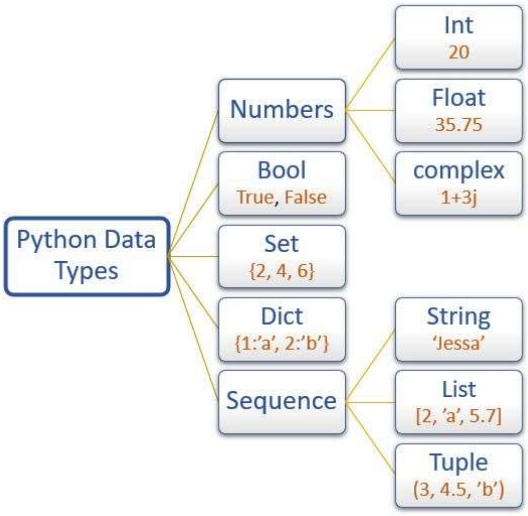
What is Python?

Python is a general-purpose, high-level programming language. Its design makes it very readable, which is more important than it sounds. Compared with other popular languages, Python has a gentle learning curve that makes the user productive sooner, yet it has depths that one can explore as they gain expertise.

Since its introduction in the 1990s, Python has become one of the most widely used programming languages in the software industry. Also, students learning their first programming language find Python the ideal tool to understand the development of computer programs.

Python fast enough for most applications, but it might not be fast enough for some of the more demanding ones. If your program spends most of its time calculating things (the technical term is CPU-bound), a program written in C, C++, C#, Java, Rust, or Go will generally run faster than its Python equivalent. But not always!

Python Datatypes



Data type	Description	Mutable / Immutable	Example
`int`	To store integer values	Immutable	`n = 20`
`float`	To store decimal values	Immutable	`n = 20.75`
`complex`	To store complex numbers (real and imaginary part)		`n = 10+20j`
`str`	To store textual/string data	Immutable	`name = 'Jessa'`
`bool`	To store boolean values	Immutable	`flag = True`
`list`	To store a sequence of mutable data	Mutable	`l = [3, 'a', 2.5]`
`tuple`	To store sequence immutable data	Immutable	`t = (2, 'b', 6.4)`
`dict`	To store key: value pair	Mutable	`d = {1:'J', 2:'E'}`
`set`	To store unorder and unindexed values	Mutable	`s = {1, 3, 5}`

`frozenset`	To store immutable version of the set	Immutable	`f_set=frozenset({5,7})`
`range`	To generate a sequence of number		`numbers = range(10)`
`bytes`	To store bytes values	Immutable	`b=bytes([5,10,15,11])`

Variables

Python, like most computer languages, lets you define variables—names for values in your computer's memory that you want to use in a program. Python variable names have some rules:

- They can contain only these characters:
  - Lowercase letters (a through z)
  - Uppercase letters (A through Z)
  - Digits (0 through 9)
  - Underscore (_)
- They are case-sensitive: thing, Thing, and THING are different names.
- They must begin with a letter or an underscore, not a digit.
- Names that begin with an underscore are treated specially.
- They cannot be one of Python's **reserved** words (also known as **keywords**).

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Operators in Python

There are 6 types of operators in Python as listed below.

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

1. Arithmetic Operators

- Python Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, and division.
- In Python 3.x the result of division is a floating-point number.

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Example

```
a = 7
b = 2

# addition
print ('Sum: ', a + b)

# subtraction
print ('Subtraction: ', a - b)

# multiplication
print ('Multiplication: ', a * b)

# division
print ('Division: ', a / b)

# floor division
print ('Floor Division: ', a // b)

# modulo
print ('Modulo: ', a % b)

# a to the power b
print ('Power: ', a ** b)
```

```
Sum:  9
Subtraction:  5
Multiplication:  14
Division:  3.5
Floor Division:  3
Modulo:  1
Power:  49
```

Precedence of Arithmetic Operators in Python

The precedence of Arithmetic Operators in Python is as follows:

1. P – Parentheses
2. E – Exponentiation
3. M – Multiplication (Multiplication and division have the same precedence)
4. D – Division
5. A – Addition (Addition and subtraction have the same precedence)
6. S – Subtraction

2. Comparison Operators

In Python Comparison of Relational operators compares the values. It either returns `True` or `False` according to the condition.

Operator	Description	Syntax
`>`	Greater than: True if the left operand is greater than the right	`x > y`
`<`	Less than: True if the left operand is less than the right	`x < y`
`==`	Equal to: True if both operands are equal	`x == y`
`!=`	Not equal to – True if operands are not equal	`x != y`
`>=`	Greater than or equal to True if the left operand is greater than or equal to the right	`x >= y`
`<=`	Less than or equal to True if the left operand is less than or equal to the right	`x <= y`

Precedence of Comparison Operators in Python

- In Python, the comparison operators have lower precedence than the arithmetic operators.
- All the operators within comparison operators have the same precedence order.

3. Logical Operators

Python Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements.

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if the operand is false	not x

Precedence of Logical Operators in Python

The precedence of Logical Operators in Python is as follows:

1. Logical not
2. logical and
3. logical or

Example

```
# Need for operator precedence

# Precedence of or & and
meal = "fruit"

money = 0

if meal == "fruit" or meal == "sandwich" and money >= 2:
    print("Lunch is ready.")
else:
    print("Can't deliver lunch.")
```

```
Lunch is ready.
```

```
# operator precedence

# Precedence of or & and
meal = "fruit"

money = 0

if (meal == "fruit" or meal == "sandwich") and money >= 2:
    print("Lunch is ready.")
else:
    print("Can't deliver lunch.")
```

```
Can't deliver lunch.
```

4. Bitwise Operators

Python Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Precedence of Bitwise Operators in Python

The precedence of Bitwise Operators in Python is as follows:

1. Bitwise NOT
2. Bitwise Shift
3. Bitwise AND
4. Bitwise XOR
5. Bitwise OR

5. Assignment Operators

Python Assignment operators are used to assign values to the variables.

Operator	Description	Syntax
=	Assign the value of the right side of the expression to the left side operand	x = y + z
+=	Add AND: Add right-side operand with left-side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent (raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a= a << b

6. Identity Operators and Membership Operators

Identity Operators

In Python, `is` and `is not` the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

```
is          True if the operands are identical
is not      True if the operands are not identical
```

Membership Operators

In Python, `in` and `not in` are the membership operators that are used to test whether a value or variable is in a sequence.

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

Ternary Operators

In Python, Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being `True` or `False`. It simply allows testing a condition in a single line replacing the multiline if-else making the code compact.

```
[on_true] if [expression] else [on_false]
```

Example

```
# Ternary operators

a = 20
b = 10

min = a if a < b else b

print(min)
```

```
10
```

The precedence order (Highest precedence from top)

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x  -x  ~x`	Unary plus, unary minus, and bitwise NOT
`*  /  //  %`	Multiplication, division, floor division, and modulus
`+  -`	Addition and subtraction
`<<  >>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==  !=  >  >=  <  <=  is  is not  in  not in`	Comparisons, identity, and membership operators
`not`	Logical NOT
`and`	AND
`or`	OR

If two operators have the same precedence, the expression is evaluated from left to right.

Python print() function

- The python `print()` function is used to print a python object(s) in Python as standard output.
- **Syntax:**

```
print(object(s), sep, end, file, flush)
```

- **Parameters:**
  - Object(s): It can be any python object(s) like string, list, tuple, etc. But before printing all objects get converted into strings.
  - `sep`: It is an optional parameter used to define the separation among different objects to be printed. By default an empty string(“”) is used as a separator.

- **end**: It is an optional parameter used to set the string that is to be printed at the end. The default value for this is set as line feed("\n").
- **file**: It is an optional parameter used when writing on or over a file. By default, it is set to produce standard output as part of **sys.stdout**.
- **flush**: It is an optional boolean parameter to set either a flushed or buffered output. If set **True**, it takes flushed else it takes buffered. By default, it is set to **False**.

Example

```
# sample python objects
list1 = [1,2,3]
tuple1 = ("A","B")
string1 = "Python is fun!!!"
int1 = 5
float1 = 3.14159
complex1 = 3+4j

# printing the objects
print(list1)
print(tuple1)
print(string1)
print(int1)
print(float1)
print(complex1)
print(list1, tuple1, string1, int1, float1, complex1)
print(list1, tuple1, string1, int1, float1, complex1, sep = '-**-')
print(list1, tuple1, string1, int1, float1, complex1, end = '-**-')
```

```
[1, 2, 3]
('A', 'B')
Python is fun!!!
5
3.14159
(3+4j)
[1, 2, 3] ('A', 'B') Python is fun!!! 5 3.14159 (3+4j)
[1, 2, 3]-**-('A', 'B')-**-Python is fun!!!-**-5-**-3.14159-**-(3+4j)
[1, 2, 3] ('A', 'B') Python is fun!!! 5 3.14159 (3+4j)-**-
```

Example

```
print('A'+'good'+'tutorial'+'on'+'python'+'print'+'function.')
print('A', 'good','tutorial','on','python','print','function.')
print('A', 'good','tutorial','on','python','print','function.',sep='\n')
```

```
Agoodtutorialonpythonprintfunction.
A good tutorial on python print function.
A
good
tutorial
on
python
print
function.
```

Example

```
str01 = 'Hello World!!!'
str02 = 'It\'s a good day.'

print('S01:', str01)
print('S02:',str02)
print('S03:',str01, end = '')
print('S04:',str02)
print('S05:',str01, str02)
print('S06:',str01 + str02)
print('S07:',str01, str02, sep = '\n')
```

```
S01: Hello World!!!
S02: It's a good day.
S03: Hello World!!!S04: It's a good day.
S05: Hello World!!! It's a good day.
S06: Hello World!!!It's a good day.
S07:
Hello World!!!
It's a good day.
```

Input Statement in Python

- The `input()` function in Python is used to take user input. By default, it returns the user input as a string
- **Example**

```
name = input("Enter your name: ")
print("Hello, " + name)
```

```
Enter your name: XYZ
Hello, XYZ
```

- **Typecasting Input**
  - If you need the input to be of a different data type, you can use typecasting.
  - For example, to take an integer input:

```
age = int(input("Enter your age: "))
print("Your age is", age)
```

```
Enter your age: 45
Your age is 45
```

  - Similarly, for floating-point numbers the float keyword can be used to typecast the variables
  -
- **Multiple Inputs**
  - You can also take multiple inputs from the user and store them in variables.

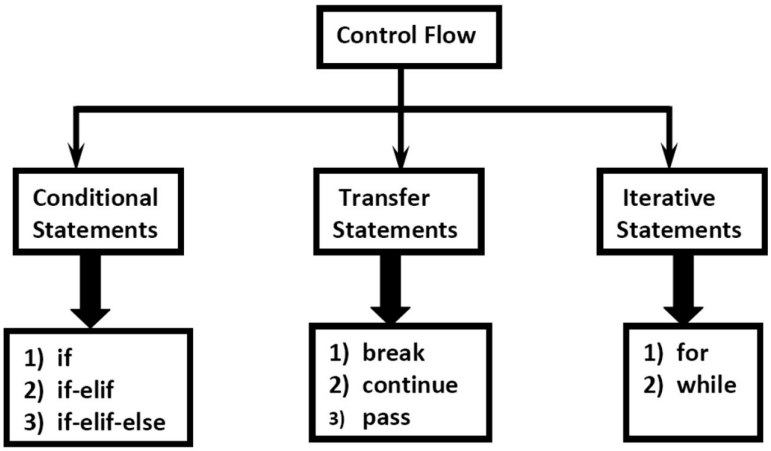
```
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
sum = num1 + num2
print("The sum is", sum)
```

```
Enter first number: 45
Enter second number: 65
The sum is 110
```

- **Important Considerations**
  - The `input()` function always returns a string. If you need another data type, you must convert it manually.
  - You can provide a prompt message to guide the user on what to input.

Flow Control Statements In Python

Flow control describes the order in which statements will be executed at runtime. Flow control statements are divided into three categories in Python.



Conditional Statements (or) Selection Statements

Based on some condition result, some group of statements will be executed and some group of statements will not be executed.

Note:

- There is no switch statement in Python. (Which is available in C and Java)
- There is no do-while loop in Python. (Which is available in C and Java)
- goto statement is also not available in Python. (Which is available in C)

i) if Statement:

In Python anywhere we are using **colon (:)** means we are defining block. The statements belonging to the block must have the same indentation level. If this is not maintained, then python throws `indentationError`.

Syntax:

```
if(condition):
      #Executes this block if the condition is true
```

Python uses indentation in both the blocks to define the scope of the code. Other programming languages often use curly brackets for this purpose.

Example

```
if 10<20:
    print('10 is less than 20')
print('End of Program')
```

```
10 is less than 20
End of Program
```

Example

```
if 10<20:
print('10 is less than 20')
print('End of Program')
```

```
  File "<ipython-input-3-f2d3b9a6180e>", line 2
    print('10 is less than 20')
        ^
IndentationError: expected an indented block
```

ii) if - else Statement:

Syntax

```
if(condition):
      #Executes this block if the condition is true
else:
   #Executes this block if the condition is false
```

Example

```
name = input('Enter Name : ')
if name == 'Karthi':
    print('Hello Karthi! Good Morning')
else:
    print('Hello Guest! Good MOrning')
print('How are you?')
```

```
Enter Name : Karthi
Hello Karthi! Good Morning
How are you?
```

```
Enter Name : ram
Hello Guest! Good MOrning
How are you?
```

iii) if-elif-else Statement:

Syntax

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    Statement
```

Example

```
# ticket price
# 1-3 year old -->free
# 4-10 year old----> 150rs
# 11-60 year old---> 250rs
# above 60 old--->200rs

age=int(input("enter your age: "))
if age==0 or age<0:
    print("invalid age please enter right age:")
elif 0<age<=3:
    print("your ticket price is FREE")
elif 3<age<=10:
     print("your ticket price is  rs 150")
elif 10<age<=60:
     print("your ticket price is rs 250")
else:
     print("your ticket price is rs 200")
```

Output

```
enter your age: 10
your ticket price is  rs 150

enter your age: 75
your ticket price is rs 200
```

iv) Nested If

Syntax

```
if (condition1):
      #Executes if condition1 is true
      if (condition2):
            #Executes if condition2 is true
      else:
            #Executes if condition2 is False
      #Condition2 ends here
else:
      #Executes if condition1 is False
#Condition1 ends here
```

Take care of the indentation to define the scope of each statement. You can have as many levels of nesting as required, but it makes the program less optimized, and as a result, can be more complex to read and understand. Therefore, you should always try to minimize the use of nested IF statements.

Example

```
c=21
if c<25:
    if c%2==0:
        print("c is an even number less than 25")
    else:
        print("c is an odd number less than 25")
else:
    print("c is greater than 25")
```

```
c is an odd number less than 25
```

```
c=50
if c<25:
    if c%2==0:
        print("c is an even number less than 25")
    else:
        print("c is an odd number less than 25")
else:
    print("c is greater than 25")
```

```
c is greater than 25
```

Iterative Statements

i) For Loop

The **for** loop in Python is used to iterate over a sequence (**list, tuple, string**) or other iterable objects. Iterating over a sequence is called traversal. If we want to execute some action for every element present in some sequence (it may be string or collection) then we should use for loop.

A **for** loop repeats a block of statements as its loop variable iterates through a sequence. The sequence can be an arithmetic progression, the items of a list or tuple, the characters of a string, or the lines of a file object.

Syntax

```
for x in sequence:
    body
```

Example:

Write a Program to print characters present in the given string.

```
s="Sahasra"
for x in s :
    print(x)
```

```
S
a
h
a
s
r
a
```

Write a Program to print characters present in string index wise.

```
s=input("Enter some String: ")
i=0
for x in s :
    print("The character present at ",i,"index is :",x)
    i=i+1
```

```
Enter some String: Karthikeya
The character present at  0 index is : K
The character present at  1 index is : a
The character present at  2 index is : r
The character present at  3 index is : t
The character present at  4 index is : h
The character present at  5 index is : i
The character present at  6 index is : k
The character present at  7 index is : e
The character present at  8 index is : y
The character present at  9 index is : a
```

Write a Program to print Hello 10 times.

```
s = 'Hello'
for i in range(10):
    print(s)
```

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

ii) while loop:

If we want to execute a group of statements iteratively until some condition false, then we should go for while loop.

Syntax:

```
while <condition>:
    body
```

Examples:

Write a Program to print numbers from 1 to 10 by using while loop.

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

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

Write a Program to display the sum of first n numbers.

```
n=int(input("Enter number:"))
sum=0
i=1
while i<=n:
    sum=sum+i
    i=i+1
print("The sum of first",n,"numbers is :",sum)
```

```
Enter number:10
The sum of first 10 numbers is : 55
```

Infinite Loops

Sometimes a while loop can execute infinite number of times without stopping also. The body of the while loop keep on executing because condition is always true, hence the program never terminates. Sometimes users do not update the value of the loop variable and hence the loop gets executed infinitely.

Example

```
i = 1
while True:         #
    print('Hello', i)
    i=i+1
```

If by mistake, our program entered into an infinite loop, we can avoid the same by using **break** statement.

Syntax:

```
while <test condition1>:
      body
      if <test condition2>:
            break
```

Nested Loops

Sometimes we can take a loop inside another loop, which are also known as nested loops. The time complexity of such loops is $O(n^2)$. Hence such loops should be avoided if possible.

Syntax

```
while <test condition1>:            # Outer Loop
      body
      while <test condition2>:      # Inner Loop
            body

for x1 in iterable1:                # Outer Loop
      body
      for x2 in iterable2:          # Inner Loop
            body
```

Example

Write a program to print hello six times using nested for loop.

```
for i in range(3):
    for j in range(2):
        print('Hello')
```

```
Hello
Hello
Hello
Hello
Hello
Hello
```

Write a program to display *'s in Right angled triangled form using nested loops.

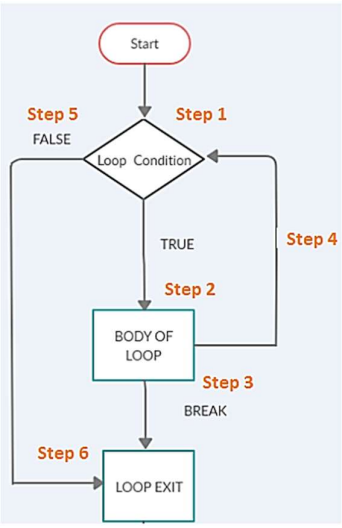
```
n = int(input("Enter number of rows:"))
for i in range(1,n+1):
    for j in range(1,i+1):
        print("*",end=" ")
    print()
```

```
Enter number of rows:7
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
```

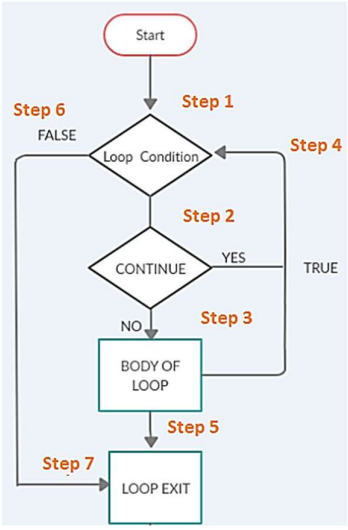
Transfer Statements

i) Break

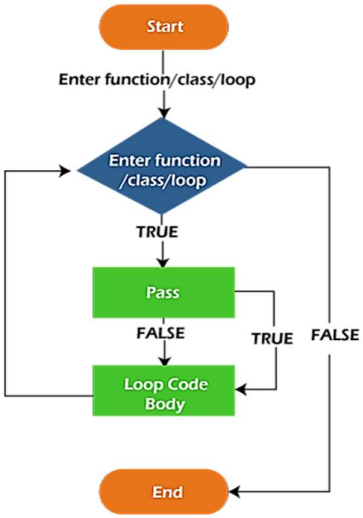
We can use break statement inside loops to break loop execution based on some condition. When the statement **break** is executed in the body of a for loop, the loop is terminated and the loop variable keeps its current value.



Break



Continue



Pass

ii) Continue

When the statement continue is **executed** in the body of a for loop, the remaining statements in the body of the loop are skipped and execution continues with the next iteration of the loop.

Both `break` and `continue` statements usually appear in the body of an if statement and provide an efficient way of transferring control.

iii) The pass Statement

The header of a for loop must be followed by an indented block of at least one statement. However, there are times when you want the loop to cycle through a sequence and not do anything. In that case, the pass statement should be used. The pass statement is a do-nothing placeholder statement.

Examples

Break Example

```
for number in range(10):
    if number == 5:
        print('Breaking....')
        break    # break here

    print('Number is ' + str(number))

print('Out of loop')
```

Output

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Breaking....
Out of loop
```

Continue Example

```
for number in range(10):
    if number == 5:
        print('Skipping this and \n Continuing to next....')
        continue    # continue here

    print('Number is ' + str(number))

print('Out of loop')
```

Output

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Skipping this and
 Continuing to next....
Number is 6
Number is 7
Number is 8
Number is 9
Out of loop
```

Pass Example

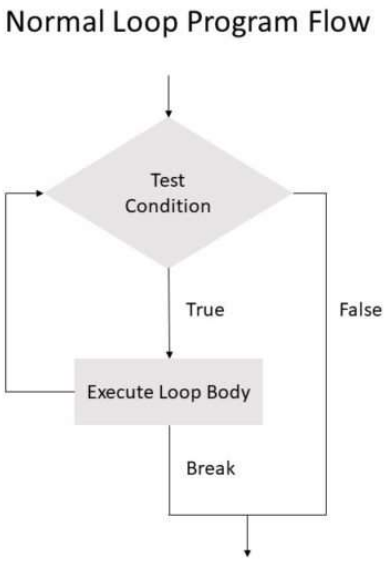
```
for number in range(10):
    if number == 5:
        print('Passing....')
        pass    # pass here

    print('Number is ' + str(number))

print('Out of loop')
```

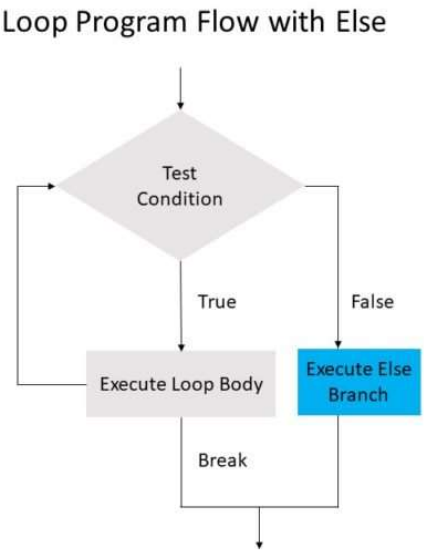
Output

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Passing....
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9
Out of loop
```

Else in Loop Structures

In most of the programming languages (C/C++, Java, etc), the use of else statement has been restricted with the if conditional statements. But Python also allows us to use the else condition with both for and while loops. The else block just after for/while is executed only when the loop is NOT terminated by a break statement.





Example

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Output

```
0
1
2
3
4
5
Finally finished!
```

Example

```
for x in range(6):
  print(x)
print("Finally finished!")
```

Output

```
0
1
2
3
4
5
Finally finished!
```

Example

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
print('==========================')
```

Output

```
0
1
2
```

Example

```
fruits=['apple', 'banana', 'mango', 'strawberry']
fruits_len = len(fruits)
index = 0
searchItem = input('Enter a fruit name: ')

while index < fruits_len:
    if fruits [index] == searchItem:
        print(searchItem, 'is available.')
        break
    index+=1
else:
    print(searchItem, 'is not available.')
print('==========================')
```

Output

```
Enter a fruit name: mango
mango is available.

Enter a fruit name: orange
orange is not available.
```

Functions In Python

Functions are used to break complex problems into small problems to be solved one at a time. Functions allow us to write and read a program in such a way that we first focus on the tasks and later on how to accomplish each task. They also eliminate repetitive code and can be reused in other programs.

There are two types of functions— those that are designed to return values and those that execute lines of code without the intent to return a value. The second type of function often displays output with **print** statements or creates a file. We will begin by discussing functions designed to return a value.

Functions in python can be of two types.

1. Built-in functions
2. User defined functions

Built-in functions

Python has many built- in functions. In one respect, functions are like miniature programs. They receive input, they process the input, and they produce output.

Built-in Functions

A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		**_**
	iter()		__import__()

User defined functions

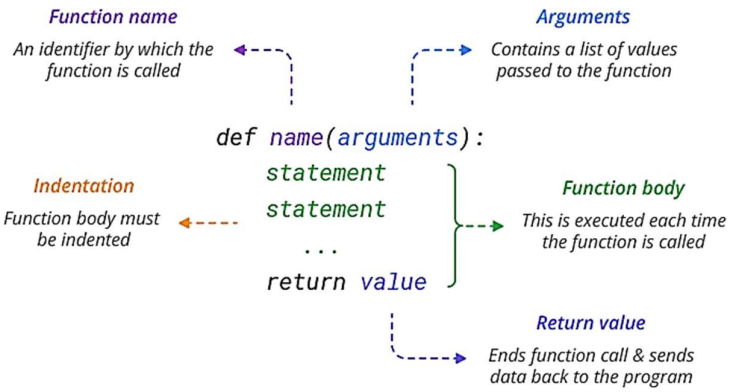
In Python, user-defined functions are functions created by the programmer using the def keyword just like in the pseudo-code below:

```
# User Defined Function
def user_defined_function(argument1, argument2, ...):
    # statement
    return [expression]
```

Syntax of a Function Declaration

The syntax of function declarations in Python contains a number of things:

- **def** keyword: Keyword used to declare a function
- function **name**: Any name to give to the function
- function parameters (optional): Parameters to be used inside the function
- colon (:): Operator is the required character to begin a block of code in Python
- function body: What the code should do.
- function **return** (optional): Keyword used to declare the return statement
- **return expression** (optional): What the code should return



Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Types of Arguments

Python handles function arguments in a very flexible manner, compared to other languages. It supports multiple types of arguments in the function definition. Here's the list:

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments

1. Positional Arguments

The most common are positional arguments, whose values are copied to their corresponding parameters in order. The only downside of positional arguments is that you need to pass arguments in the order in which they are defined.

Example	Output

```
def func(name, job):
    print(name, 'is a', job, '.')

func('Bob', 'developer')
func('developer', 'Bob')
```

```
Bob is a developer .
developer is a Bob .
```

2. Keyword Arguments

To avoid positional argument confusion, you can pass arguments using the names of their corresponding parameters. In this case, the order of the arguments no longer matters because arguments are matched by name, not by position.

Example

```
# Keyword arguments can be put in any order
def func(name, job):
    print(name, 'is a', job)

func(name='Bob', job='developer')
func(job='developer', name='Bob')
```

Output

```
Bob is a developer
Bob is a developer
```

3. Default Arguments

You can specify default values for arguments when defining a function. The default value is used if the function is called without a corresponding argument.

Example

```
# Set default value 'developer' to a 'job' parameter
def func(name, job='developer'):
    print(name, 'is a', job)

func('Bob', 'manager')
func('Bob')
```

Output

```
Bob is a manager
Bob is a developer
```

How to Use a Python Function

To use a function in Python, call the function by its name with parentheses containing the required arguments of the function. Additionally, any optional argument can be added to the function.

```
function_name(required_arguments, optional_arguments)
```

Investigate How to use a Function

Pass the function name to the built-in the **help()** function to investigate what a function does and how to use it.

```
# Investigate a function
help(sum)
```

Docstring

The first string after the function is called the Document string or Docstring in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

The below syntax can be used to print out the docstring of a function.

```
print(function_name.__doc__)
```

Example

```
def evenOdd(x):
    """Function to check if the number is even or odd"""

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


# Driver code to call the function
print(evenOdd.__doc__)
```

Output

```
Function to check if the number is even or odd
```

Python Function within Functions (Inner Functions / Nested Functions)

A function that is defined inside another function is known as the **inner function** or **nested function**. Nested functions can access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function.

Example

```
# Python program to
# demonstrate accessing of
# variables of nested functions

def f1():
    s = 'Learning Python is fun.'

    def f2():
        print(s)

    f2()

# Driver's code
f1()
```

Output

```
Learning Python is fun.
```

Reasons to use nested functions

While there are many valid reasons to use nested functions, among the most common are encapsulation and closures / factory functions.

- **Data encapsulation**
  There are times when you want to prevent a function or the data it has access to from being accessed from other parts of your code, so you can *encapsulate* it within another function. When you nest a function like this, it's hidden from the global scope. Because of this behaviour, data encapsulation is sometimes referred to as data hiding or data privacy.

Example

```
def outer():
  print("I'm the outer function.")
  def inner():
    print("And I'm the inner function.")
  inner()

inner()
```

Output

```
Traceback (most recent call last):
  File "C:/Users/G R Patra/OneDrive/CM_FDP/AIML/MLW Course 202
4/Programs/demoNestedFucn.py", line 23, in <module>
    inner()
NameError: name 'inner' is not defined. Did you mean: 'iter'?
```

In the code above, the inner function is only available from within the function outer. If you try to call inner from outside the function, you'll get the error above. The outer function can be called without any issues.

Example	Output

```
def outer():
  print("I'm the outer function.")
  def inner():
    print("And I'm the inner function.")
  inner()

outer()
```

```
I'm the outer function.
And I'm the inner function.
```

- **Closures**
  But what would happen if the outer function returns the inner function itself, rather than calling it like in the example above? In that case you would have what's known as a **closure**.

  The following are the conditions that are required to be met to create a closure in Python:
  1. *There must be a nested function*
  2. *The inner function must refer to a value that is defined in the enclosing scope*
  3. *The enclosing function must return the nested function*

Example	Output

```
def num1(x):
  def num2(y):
    return x + y
  return num2

print('Result =', num1(10)(5))
```

```
Result = 15
```

Lambda Functions

Python lambdas are little, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned.

For example, the following python function can be written as a lambda function as follows.

```
def add_one(x):
    return x + 1

print(add_one(5))
```

```
add_oneL = lambda x: x + 1

print(add_oneL(5))
```

Example

```
full_name = lambda first,middle,last: f'Full name: {first} {middle} {last}'

print(full_name('Gyana', 'Ranjan','Patra'))
```

Output

```
Full name: Gyana Ranjan Patra
```

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Output

```
22
33
```

Lambda function characteristics

- Anonymous (no name) and typically used for short, simple functions.
- A lambda function can't contain any statements. (Example: `assert x == 2`)
- In contrast to a normal function, a Python lambda function is a single expression.
- Like a normal function object defined with def, Python lambda expressions support all the different ways of passing arguments.

The main function

In python programming a main function may be declared. The main function needs to be called at the end of the program which initiates the task to be performed.

```
1  def func1(parameters):
2      statements
3  def func2(parameters):
4      statements
5  def func3(parameters):
6      statements
7      .
8      .
9      .
10 def main():
11     statements
12
13
14 if __name__ = '__main__':
15     main
```

The statement in line 14 ensures that the main function gets called from the file where it has been written. It makes sure that the main function of this program is not accidentally executed when this has been imported to another program. Henceforward all programs need to be written in this way.

Strings

Sentences, phrases, words, letters of the alphabet, names, telephone numbers, addresses, and social security numbers are all examples of strings.

Strings in Python are "**immutable**" which means they cannot be changed after they are created. Some other immutable data types are integers, float, boolean, etc.

The immutability of Python string is very useful as it helps in hashing, performance optimization, safety, ease of use, etc.

Example	Output

`my_string = "Hello, world!"`
 	
`# Attempt to modify the string`
`my_string[0] = 'h'`
	`Traceback (most recent call last):`
	`  File "C:/Users/G R Patra/OneDrive/CM_FDP/AIML/MLW Course 2024/Programs/stringIndexing.py", line 4, in <module>`
	`  my_string[0] = 'h'`
	`TypeError: 'str' object does not support item assignment`

String Literals

- A string literal is a sequence of characters that is treated as a single item.
- The characters in strings can be any characters found on the keyboard (such as letters, digits, punctuation marks, and spaces) and many other special characters.
- String literals are written as a sequence of characters surrounded by either single quotes (') or double quotes (").
- Opening and closing quotation marks must be the same type— either both double quotes or both single quotes.

Examples

```
"John Doe"
'5th Avenue'
```

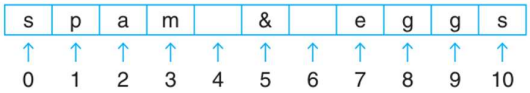
Variables

- Variables also can be assigned string values.
- As with variables assigned numeric values, variables assigned string values are created (that is, come into existence) the first time they appear in assignment statements.

Indices and Slices

- In Python, the position or index of a character in a string is identified with one of the numbers 0, 1, 2, 3, ...
- For instance, the first character of a string is said to have index 0, the second character is said to have index 1, and so on.
- If **str1** is a string variable or literal, then **str1[i]** is the character of the string having index **i**.

Examples



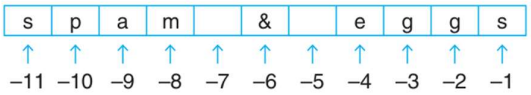
- A **substring** or **slice** of a string is a sequence of consecutive characters from the string.
  - If **str1** is a string, then **str1[m:n]** is the substring beginning at position **m** and ending at position **n - 1**.

- If **m≥n**, that is, if the character in position **m** is not to the left of the character in position **n**, then the value of **str1[m:n]** will be the empty string (" "), the string with no characters.

Negative Indices

- The indices discussed above specify positions from the left side of the string.
- Python also allows strings to be indexed by their position with regards to the right side of the string by using negative numbers for indices.
- With negative indexing, the rightmost character is assigned index -1, the character to its left is assigned index -2, and so on.

Example



Example	Output

```
print('Positive indexing in Strings')
print('============================')

print("Python")
print("Python"[1], "Python"[5], "Python"[2:4])
print('============================')


str0 = "Python"
print(str0)
print(str0[1], str0[5], str0[2:4])
print('============================')

str1 = "Hello World!"
print(str1.find('W'))
print(str1.find('x'))
print(str1.rfind('l'))

print('============================')
print('Negative indexing in Strings')
print('============================')
print("Python")
print("Python"[-1], "Python"[-4], "Python"[-5:-2])


str1 = "spam & eggs"
print(str1[-1])
print(str1[-8:-3])
print(str1[0:-1])
```

```
Positive indexing in Strings
============================
Python
y n th
============================
Python
y n th
============================
6
-1
9

============================
Negative indexing in Strings
============================
Python
n t yth
s
m & e
spam & egg
```

Default bounds for Slices

- **str1[:n]** – consists of all the characters from the beginning of the string to **str1[n- 1]**
- **str1[m:]** – consists of all the characters from **str1[m]** to the end of the string
- **str1[:]** – consists of the entire string **str1**.

Example	Output

```
print('============================')
print('Default bounds for Slices')
print('============================')
print("Python"[2:])
print("Python"[:4])
print("Python"[:])
print()
print("Python"[-3:])
print("Python"[:-3])
```

```
============================
Default bounds for Slices
============================
thon
Pyth
Python

hon
Pyt
```

String Concatenation

```
str3 = str1 + str2
```

String Repetition

```
str2 = IntegerValue*str1
```

String Functions and methods

	Method	Description
1.	capitalize()	Converts the first character to upper case
2.	casefold()	Converts string into lower case
3.	center()	Returns a centered string
4.	count()	Returns the number of times a specified value occurs in a string
5.	encode()	Returns an encoded version of the string
6.	endswith()	Returns true if the string ends with the specified value
7.	expandtabs()	Sets the tab size of the string
8.	find()	Searches the string for a specified value and returns the position
9.	format()	Formats specified values in a string
10.	format_map()	Formats specified values in a string
11.	index()	Searches the string for a specified value and returns the position
12.	isalnum()	Returns True if all characters in the string are alphanumeric
13.	isalpha()	Returns True if all characters in the string are in the alphabet
14.	isascii()	Returns True if all characters in the string are ascii characters
15.	isdecimal()	Returns True if all characters in the string are decimals
16.	isdigit()	Returns True if all characters in the string are digits
17.	isidentifier()	Returns True if the string is an identifier
18.	islower()	Returns True if all characters in the string are lower case
19.	isnumeric()	Returns True if all characters in the string are numeric
20.	isprintable()	Returns True if all characters in the string are printable
21.	isspace()	Returns True if all characters in the string are whitespaces
22.	istitle()	Returns True if the string follows the rules of a title
23.	isupper()	Returns True if all characters in the string are upper case
24.	join()	Converts the elements of an iterable into a string
25.	ljust()	Returns a left justified version of the string
26.	lower()	Converts a string into lower case
27.	lstrip()	Returns a left trim version of the string
28.	maketrans()	Returns a translation table to be used in translations
29.	partition()	Returns a tuple where the string is parted into three parts
30.	replace()	Returns a string where a specified value is replaced with a specified value
31.	rfind()	Searches the string for a specified value and returns the last position
32.	rindex()	Searches the string for a specified value and returns the last position
33.	rjust()	Returns a right justified version of the string

34.	`rpartition()`	Returns a tuple where the string is parted into three parts
35.	`rsplit()`	Splits the string at the specified separator, and returns a list
36.	`rstrip()`	Returns a right trim version of the string
37.	`split()`	Splits the string at the specified separator, and returns a list
38.	`splitlines()`	Splits the string at line breaks and returns a list
39.	`startswith()`	Returns true if the string starts with the specified value
40.	`strip()`	Returns a trimmed version of the string
41.	`swapcase()`	Swaps cases, lower case becomes upper case and vice versa
42.	`title()`	Converts the first character of each word to upper case
43.	`translate()`	Returns a translated string
44.	`upper()`	Converts a string into upper case
45.	`zfill()`	Fills the string with a specified number of 0 values at the beginning

Chained methods

- Consider the following two lines of code:

```
praise = "Good Doggie".upper()
numberOfGees = praise.count('G')
```

- These two lines can be combined into the single line below that is said to chain the two methods.

```
numberOfGees = "Good Doggie".upper().count('G')
```

- Chained methods are executed from left to right. Chaining often produces clearer code since it eliminates temporary variables, such as the variable `praise` above.

The int, float, eval, and str Functions

- If **str1** is a string containing a whole number, the **int** function will convert the string to an integer.
- If **str1** is a string containing any number, the **float** function will convert the string to a floating- point number.
- The **float** function also converts an integer to a floating-point number.
- If **str1** is a string consisting of a numeric expression, the **eval** function will evaluate the expression to an integer or floating-point number as appropriate.
- If x is an integer, the value of **int(x)** is x.
- If x is a floating- point number, the **int** function removes the decimal part of the number.
- The **float** function operates as expected.
- The **eval** function cannot be applied to numeric literals, variables, or expressions.
- A string cannot be concatenated with a number. However, we can use the **str** function to convert the number into a string and then concatenate both.

The input Function

- The general form of an **input** statement is

```
variableName = input(prompt)
```

where `prompt` is a string that requests a response from the user.

- The **input** function always returns a string. However, a combination of an input function and an **int**, **float**, or **eval** function allows numbers to be input into a program.

Line Continuation

- A long statement can be split across two or more lines by ending each line (except the last) with a backslash character (`\`). For instance, the line

```
quotation = "Well written code is its own best documentation."
```
can be written as
```
quotation = "Well written code is its own " + \
            "best documentation."
```

- Any code enclosed in a pair of parentheses can also span multiple lines.
```
quotation = ("Well written code is its own " +
                  "best documentation.")
```

Indexing and Slicing Out of bounds

- Python does not allow out of bounds indexing for individual characters of strings but does allow out of bounds indices for slices.
- For instance, if
```
str1 = "Python"
```
- then `print(str1[7])` and `print(str1[-7])` trigger the Traceback error message `IndexError`.
- If the left index in a slice is too far negative, the slice will start at the beginning of the string,
- Similarly, if the right index is too large, the slice will go to the end of the string.
- For instance,
```
str1[-10:10] is "Python"      # equivalent to str1[:] or str1[0:7]
str1[-10:3] is "Pyt"          # equivalent to str1[:3]
str1[2:10] is "thon"          # equivalent to str1[2:]
```

F-Strings

- Field widths are to format text in a specified number of character positions.
- By default, Python
  - right0-aligns numbers and
  - left-aligns other values such as strings
- Python formats float values with six digits of precision to the right of the decimal point by default. Unused character positions are filled by blanks.
```
In [1]: f'[{27:10d}]'
Out[1]: '[        27]'

In [2]: f'[{3.5:10f}]'
Out[2]: '[  3.500000]'

In [3]: f'[{"hello":10}]'
Out[3]: '[hello     ]'
```
- We can specify left and right alignment with < and >.
```
In [4]: f'[{27:<15d}]'
Out[4]: '[27              ]'

In [5]: f'[{3.5:<15f}]'
Out[5]: '[3.500000        ]'

In [6]: f'[{"hello":>15}]'
Out[6]: '[          hello]'
```

- We can center the values as follows.

```
In [7]: f'[{27:^7d}]'
Out[7]: '[  27   ]'

In [8]: f'[{3.5:^7.1f}]'
Out[8]: '[  3.5  ]'

In [9]: f'[{"hello":^7}]'
Out[9]: '[ hello ]'
```

- We can format numbers with thousands separators by using a comma (,), as follows.

```
In [4]: f'{12345678:,d}'
Out[4]: '12,345,678'

In [5]: f'{123456.78:,.2f}'
Out[5]: '123,456.78'
```

- We can format floating point numbers in two ways as follows.

```
value = 17.489
print('{:.2f}'.format(17.489))          17.49
print(f'{value:0.2f}')                  17.49
```

- A format string may contain multiple placeholders, in which case the format method's arguments correspond to the placeholders from left to right.

```
first = 'Amanda'
last = 'Blackwell'
print('{} {}'.format(first, last))      Amanda Blackwell
print(f'{first} {last}')                Amanda Blackwell
```

Practice Exercise 01

Run the following codes and find the output.

```
poem = '''All that doth flow we cannot liquid name
Or else would fire and water be the same;
But that is liquid which is moist and wet
Fire that property can never get.
Then 'tis not cold that doth the fire put out
But 'tis the wet that makes it die, no doubt.'''

print(poem)
print('=======================================')
print('Q: Get the first 13 characters:')
print(poem[:13])
print('---------------------------------------')

print('Q: Get the last 13 characters:')
print(poem[-13:])
print('---------------------------------------')

print('Q: How many characters are in this poem?')
print(len(poem))
print('---------------------------------------')

print('Q: Does it start with the letters All?')
print(poem.startswith('All'))
print('---------------------------------------')

print('Q: Does it end with That\'s all, folks!?')
print(poem.endswith('That\'s all, folks!'))
print('---------------------------------------')

print('Q: find the offset of the first '+\
      'occurrence of the word the in the poem.')

word = 'the'
print('position: ',poem.find(word))
print('position: ', poem.index(word))
print('---------------------------------------')

print('Q: find the offset of the last '+\
      'occurrence of the word the in the poem.')

word = 'the'
print('position: ',poem.rfind(word))
print('position: ', poem.rindex(word))
print('---------------------------------------')

print('Q: How many times does the three-letter '+\
       'sequence the occur?')
print(poem.count(word))
print('---------------------------------------')
```

```
print('Q: Are all of the characters in the poem '+\
      'either letters or numbers?')
print(poem.isalnum())
print('----------------------------------------')
```

Practice Exercise 02

Run the following codes and find the output.

```
setup = 'a duck goes into a bar...'
print(setup)

## Change Case
print('========================================')
print('Q: Remove . sequences from both ends:')
print(setup.strip('.'))
print('----------------------------------------')

print('Q: Capitalize the first word:')
print(setup.capitalize())
print('----------------------------------------')

print('Q: Capitalize all the words:')
print(setup.title())
print('----------------------------------------')

print('Q: Convert all characters to uppercase:')
print(setup.upper())
print('----------------------------------------')

print('Q: Convert all characters to lowercase:')
print(setup.lower())
print('----------------------------------------')

print('Q: Swap uppercase and lowercase:')
print(setup.swapcase())
print('----------------------------------------')

print('Q: Swap uppercase and lowercase:')
print(setup.swapcase())
print('----------------------------------------')

## Alignment
print('Q: Center the string within 30 spaces:')
print(setup.center(30))
print('----------------------------------------')

print('Q: Left justify:')
print(setup.ljust(30))
print('----------------------------------------')

print('Q: Right justify:')
```

```
print(setup.rjust(30))
print('----------------------------------------')

## Formatting with f-string
thing = 'duck'
place = 'pond'
print(f'The {thing} is in the {place}.')
print(f'The {thing.capitalize()} is '+
      f'in the {place.rjust(20)}')
```

Lists, Tuple, Dictionary, Set

- Python provides us with lists, tuples, dictionaries and set, all of which have become synonym with ease of programming and can be used in diverse applications.
- In Python,
  - A **list** can contain different types of elements and is mutable.
  - A **tuple** can contain different types of elements and is immutable.
  - A **dictionary**, the items can be accessed using strings as indices.
  - A **set** is a collection which is *unordered*, *immutable*, and *unindexed*. Sets are used to store multiple items in a single variable.

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[] or list()`	`[5.7, 4, 'Yes', 5.7]`
Tuple	Yes	N0	`() or tuple()`	`(5.7, 4, 'Yes', 5.7)`
Dictionary	No	Yes	`{} or dict()`	`{'June':30, 'July':31}`
Set	No	Yes	`{} or set()`	`{5.7, 4, 'Yes'}`

Lists

- Python provides a type called a list that stores a sequential collection of elements.
- Lists are good for keeping track of things by their order, especially when the order and contents might change.
- Unlike strings, lists are mutable. You can change a list in place, add new elements, and delete or replace existing elements.
- The same value can occur more than once in a list.

List Creation

The list class defines lists. To create a list, you can use list's constructor, as follows:

```
list1 = list()                # Create an empty list
list2 = list([2, 3, 4])       # Create a list with elements 2, 3, 4
list3 = list(["red", "green", "blue"])     # Create a list with strings
list4 = list(range(3, 6))     # Create a list with elements 3, 4, 5
list5 = list('abcd')      # Create a list with characters 'a','b','c','d'
```

You can also create a list by using the following syntax, which is a little simpler:

```
list1 = []                    # Same as list()
list2 = [2, 3, 4]             # Same as list([2, 3, 4])
list3 = ["red", "green"]      # Same as list(["red", "green"])
list4 = [2, "three", 4]       # Create a list of heterogeneous values
list5 = ['a', 'b', 'c', 'd']  # Same as list('abcd')
```

The elements in a list are separated by commas and are enclosed by a pair of brackets ([ ]).

Accessing Values in Lists

- Similar to strings, lists can also be sliced and concatenated.
- To access values in lists, square brackets are used to slice along with the index or indices to get value stored at that index.

Example

```
num_list = [12,43,23,65,34,87,34,65,42,45]
print("num_list: \n", num_list)
print("First element:\n", num_list[0])
print("Elements 2 to 5:\n", num_list[2:5])
print("Even indexed elements: \n", num_list[::2])
print("Odd indexed elements: \n", num_list[1::2])
print("Reversed List:\n", num_list[::-1])
```

Output

```
num_list:
 [12, 43, 23, 65, 34, 87, 34, 65, 42, 45]
First element:
 12
Elements 2 to 5:
 [23, 65, 34]
Even indexed elements:
 [12, 23, 34, 34, 42]
Odd indexed elements:
 [43, 65, 87, 65, 45]
Reversed List:
 [45, 42, 65, 34, 87, 34, 65, 23, 43, 12]
```

Deleting Values in Lists

- Items in a list can also be deleted by assigning an empty list to a slice of elements as shown below.

Example

```
list1 = list('PROGRAM')
print('list1=', list1)
list1[2:5] = []
print('After deletion: ')
print('list1=', list1)
```

Output

```
list1= ['P', 'R', 'O', 'G', 'R', 'A', 'M']
After deletion:
list1= ['P', 'R', 'A', 'M']
```

- Elements of the list or the complete list can be deleted by using `del` method.

Example

```
num_list = [1,2,3,4,5,6,7,8]     # a list is defined
print('Original List: \n', num_list)
del num_list[2:4]
print('List with deleted elements: \n',num_list)
                  # deletes numbers at index 2 and 3

del num_list
print('Printing after deleting the list.')
print(num_list)
```

Output

```
Original List:
 [1, 2, 3, 4, 5, 6, 7, 8]
List with deleted elements:
 [1, 2, 5, 6, 7, 8]
Printing after deleting the list.
Traceback (most recent call last):
  File "C:/Users/gyana/OneDrive/CM_FDP/
AIML/MLW Course 2024/Programs/listDemo.
py", line 58, in <module>
    print(num_list)
NameError: name 'num_list' is not defin
ed. Did you mean: 'num_list1'?
```

Updating Values in Lists

- Once created, one or more elements of a list can be easily updated by giving the slice on the left-hand side of the assignment operator.
- You can also append new values in the list using the `append()` method.

Example

```
num_list = [4,3,6,5,7]
print("num_list: \n", num_list)
num_list[3] = 22
print("List after updation:\n", num_list)
num_list.append(33)
print("List after appending:\n", num_list)
del num_list[2]
print("List after deleting:\n", num_list)
```

Output

```
num_list:
 [4, 3, 6, 5, 7]
List after updation:
 [4, 3, 6, 22, 7]
List after appending:
 [4, 3, 6, 22, 7, 33]
List after deleting:
 [4, 3, 22, 7, 33]
```

Nested Lists

- Nested list means a list within another list.
- A list can contain elements of different data types which can include even a list.

Example

```
num_list1 = [4,3,6]
print("num_list: \n", num_list1)
nested_list = ['a', 'd', num_list1, 't', 'k', '5']
print("Nested list: \n", nested_list)
print("Nested list item [0]:", nested_list[0])
print("Nested list item [1]:", nested_list[1])
print("Nested list item [2]:", nested_list[2])
print("Nested list item [3]:", nested_list[3])
print("Nested list item [4]:", nested_list[4])
print()
print("Nested list item [2][0]:", nested_list[2][0])
print("Nested list item [2][1]:", nested_list[2][1])
print("Nested list item [2][2]:", nested_list[2][2])
```

Output

```
num_list:
 [4, 3, 6]
Nested list:
 ['a', 'd', [4, 3, 6], 't', 'k', '5']
Nested list item [0]: a
Nested list item [1]: d
Nested list item [2]: [4, 3, 6]
Nested list item [3]: t
Nested list item [4]: k

Nested list item [2][0]: 4
Nested list item [2][1]: 3
Nested list item [2][2]: 6
```

Sequence Operations in Lists & Strings

Operation	Description	Example	Output
`x in s`	`True` if element `x` is in sequence `s`.	`'a' in ['a','e','i','o','u']`	`True`
`x not in s`	`True` if element `x` is not in sequence `s`.	`'x' not in ['a','e','i','o','u']`	`True`
`s1 + s2`	Concatenates two sequences `s1` and `s2`.	`[1,2,3,4] + [5,6,7,8]`	`[1,2,3,4,5,6,7,8]`
`s * n`, or `n * s`	`n` copies of sequence `s` concatenated.	`2*[1,2,3,4]`	`[1, 2, 3, 4, 1, 2, 3, 4]`
`s[i]`	i^{th} element in sequence `s`.	`s = [1,2,3,4]` `s[1]`	`2`
`s[i : j]`	Slice of sequence `s` from index `i` to `j-1`	`s = [1,2,3,4]` `s[1:4]`	`[2,3,4]`
`len(s)`	Length of sequence `s`, i.e., the number of elements in `s`.	`s = [1,2,3,4]` `len(s)`	`4`
`min(s)`	Smallest element in sequence `s`. (does not work for heterogenous lists)	`s = [1,2,3,4]` `min(s)`	`1`
		`y = ['a','b','c','d']` `min(y)`	`'a'`
`max(s)`	Largest element in sequence `s`. (does not work for heterogenous lists)	`s = [1,2,3,4]` `max(s)`	`4`
		`y = ['a','b','c','d']` `max(y)`	`'d'`
`sum(s)`		`s = [1,2,3,4]` `sum(s)`	`10`

	Sum of all numbers in sequence `s`. (only works for numeric lists)	`y = ['a','b','c','d']` `sum(y)`	`TypeError`
`for loop`	Traverses elements from left to right in a `for` loop.		
`<, <=, >, >=, =, !=`	Compares two sequences.		
`all()`	Returns `True` if all elements of the list are true (non-zero) (or if the list is empty)	`num_list1 = [1,2,3,4]` `print(all(num_list1))`	`True`
		`num_list2 = [0,2,3,4]` `print(all(num_list2))`	`False`
`any()`	Returns `True` if any element of the list is true. If the list is empty, returns `False`	`num list = [1,2,3,4]` `print(any(num_list))`	`True`
`list()`	Converts an iterable (tuple. string, set, dictionary) to a list	`list1 = 'HELLO'` `print(list1)`	`['H','E','L','L','O']`

`sorted()`	Returns a new sorted list. The original list is not sorted.	`list1 = [3,4,2,7,5,8]`
		`list2 = sorted(listl)`
		`print(list2)`
			`[2, 3, 4, 5, 7, 8]`

Python List Operations

Python lists are versatile and provide a range of methods to perform operations on list items. Here's a quick overview of some common list methods with examples

- **Adding Items**

  `append()`: Adds an element to the end of the list.

```
fruits = ['apple', 'banana']
fruits.append('cherry')
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

  `extend()`: Extends the list by adding all elements from an iterable.

```
fruits = ['apple', 'banana']
fruits.extend(['cherry', 'date'])
print(fruits) # Output: ['apple', 'banana', 'cherry', 'date']
```

  `insert()`: Inserts an element at a specified position.

```
fruits = ['apple', 'banana']
fruits.insert(1, 'cherry')
print(fruits) # Output: ['apple', 'cherry', 'banana']
```

- **Removing Items**

  `remove()`: Removes the first occurrence of an element with the specified value.

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove('banana')
print(fruits) # Output: ['apple', 'cherry']
```

  `pop()`: Removes the element at the specified position and returns it.

```
fruits = ['apple', 'banana', 'cherry']
popped_fruit = fruits.pop(1)
print(popped_fruit) # Output: banana
```

  `clear()`: Removes all elements from the list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.clear()
print(fruits) # Output: []
```

- **Other Operations**

  `index()`: Returns the index of the first element with the specified value.

```
fruits = ['apple', 'banana', 'cherry']
index_of_banana = fruits.index('banana')
print(index_of_banana) # Output: 1
```

  `count()`: Returns the number of elements with the specified value.

```
fruits = ['apple', 'banana', 'cherry', 'banana']
count_of_banana = fruits.count('banana')
print(count_of_banana) # Output: 2
```

  `sort()`: Sorts the list in ascending order by default, uses ASCII values to sort the list.

```
fruits = ['cherry', 'banana', 'apple']
fruits.sort()
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

  `reverse()`: Reverses the order of the list.

```
fruits = ['cherry', 'banana', 'apple']
fruits.reverse()
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

  `copy()`: Returns a shallow copy of the list.

```
fruits = ['apple', 'banana', 'cherry']
fruits_copy = fruits.copy()
print(fruits_copy) # Output: ['apple', 'banana', 'cherry']
```

Key points to remember

- `insert()`, `remove()`, and `sort()` methods only modify the list and do not return any value.
- If you print the return values of these methods, you will get `None`.
- This is a design principle that is applicable to all mutable data structures in Python. The code given below illustrates this point.

Example	Output

```
num_list = [100, 200, 300, 400]
print(num_list.insert(2, 250))
```

```
None
```

- When one list is assigned to another list using the assignment operator then a new copy of the list is not made. Instead, assignment makes the two variables point to the one list in memory. This is also known as *aliasing*.
- Any change of made to one of the lists makes corresponding changes to the other list.

Example 1

```
num_list1 = [1,2,3,4,5]
num_list2 = num_list1
print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))


print('\nAfter modification of list1:')
num_list1.insert(2, 10)

print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))
```

Output

```
list1: [1, 2, 3, 4, 5]
list2: [1, 2, 3, 4, 5]
ID of list1: 1488365621952
ID of list2: 1488365621952


After modification of list1:
list1: [1, 2, 10, 3, 4, 5]
list2: [1, 2, 10, 3, 4, 5]
ID of list1: 1488365621952
ID of list2: 1488365621952
```

- To avoid the above one may use the `copy()` method or the list cloning using slicing.

Example 1

```
num_list1 = [1,2,3,4,5]
num_list2 = num_list1.copy()
print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))


print('\nAfter modification of list1:')
num_list1.insert(2, 10)

print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))
```

Output

```
list1: [1, 2, 3, 4, 5]
list2: [1, 2, 3, 4, 5]
ID of list1: 1784061850688
ID of list2: 1784064120576


After modification of list1:
list1: [1, 2, 10, 3, 4, 5]
list2: [1, 2, 3, 4, 5]
ID of list1: 1784061850688
ID of list2: 1784064120576
```

Example 2

```
num_list1 = [4,5,6,7,8]
num_list2 = num_list1[:]
print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))
print()


print('\nAfter modification of list1:')
num_list1.insert(2, 10)

print('list1:',num_list1)
print('list2:',num_list2)
```

Output

```
list1: [4, 5, 6, 7, 8]
list2: [4, 5, 6, 7, 8]
ID of list1: 2731107623040
ID of list2: 2731107489280


After modification of list1:
list1: [4, 5, 10, 6, 7, 8]
list2: [4, 5, 6, 7, 8]
ID of list1: 2731107623040
ID of list2: 2731107489280
```

Shallow Copy and Deep Copy

- In Python, we use = operator to create a copy of an object.
- You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.
- Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

- Shallow Copy
- Deep Copy

- To make these copy work, we use the copy module.

Shallow Copy

- A shallow copy creates a new object which stores the reference of the original elements.
- So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects.
- This means, a copy process does not recurse or create copies of nested objects itself.

Example

```
import copy
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)

print("Old list:\n", old_list)
print("New list (shallow copy):\n", new_list)

old_list.append([4, 4, 4])
print("Old list (changed):\n", old_list)
print("New list:\n", new_list)

old_list[1][1] = 'AA'
print("Old list:", old_list)
print("New list:", new_list)
```

Output

```
Old list:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
New list (shallow copy):
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Old list (changed):
 [[1, 2, 3], [4, 5, 6], [7, 8, 9], [4, 4, 4]]
New list:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Old list: [[1, 2, 3], [4, 'AA', 6], [7, 8, 9], [4, 4, 4]]
New list: [[1, 2, 3], [4, 'AA', 6], [7, 8, 9]]
```

- In the above program, we made changes to `old_list` i.e., `old_list[1][1] = 'AA'`. Both sublists of `old_list` and `new_list` at `index [1][1]` were modified. This is because, both lists share the reference of same nested objects.

Deep Copy

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.

Example

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

print("Old list:\n", old_list)
print("New list (deep copy):", new_list)
print()
old_list[1][0] = 'BB'
print("Old list (updated):", old_list)
print("New list:", new_list)
```

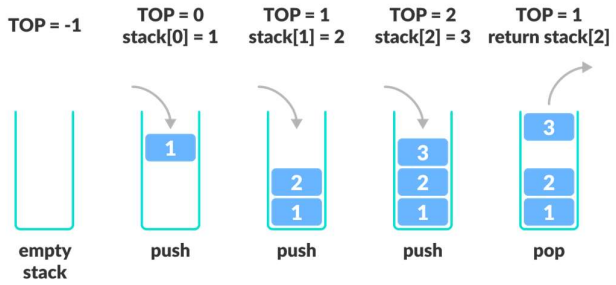
Output

```
Old list:
 [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list (deep copy): [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

Old list (updated): [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to `old_list`, we can see only the `old_list` is modified. This means, both the `old_list` and the `new_list` are independent. This is because the `old_list` was recursively copied, which is true for all its nested objects.

List as Stack

- A stack is a fundamental data structure in computer science that follows the Last In, First Out (LIFO) principle. In simpler terms, the last element added to the stack is the first one to be removed.



- The basic operations associated with a stack include:
  - **Push**: Adding an element to the top of the stack.
  - **Pop**: Removing the element from the top of the stack.
  - **Peek (or Top)**: Viewing the element at the top of the stack without removing it.
  - **IsEmpty**: Checking if the stack is empty.
- **Push Operation Using List in Python**
  - A stack is initialized as an empty list, and elements are successively added to the top of the stack using the `append` method.
- **Pop Operation Using List in Python**
  - The stack is manipulated using the `pop` method, which removes and returns the last element.
- **Top Operation Using List in Python**
  - The top element of the stack is accessed using indexing `stack[-1]` that points to the last element (Top of Stack).

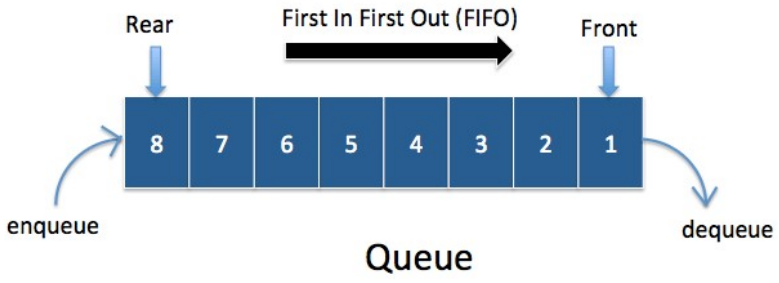
Example	Output

```
# Create an empty stack
stack = []

# Push elements onto the stack
stack.append(21)
stack.append(32)
stack.append(43)
print('Stack:\n', stack)

# Display the top of the stack
print('Top of Stack:\n', stack[-1])

# insert an element
stack.append(54)

# Display the stack
print('Stack (after push):\n', stack)

# element inserted at the back
print('Top of Stack after '+\
      'inserting an element:\n', stack[-1])

popped_element = stack.pop()
print('Popped Element:\n', popped_element)

# Display the updated stack
print('Stack (after pop):\n', stack)
```

```
Stack:
 [21, 32, 43]
Top of Stack:
 43
Stack (after push):
 [21, 32, 43, 54]
Top of Stack after inserting an element:
 54
Popped Element:
 54
Stack (after pop):
 [21, 32, 43]
```

List as Queue

- A queue is another essential data structure that follows the **First In, First Out (FIFO)** principle.
- In a queue, the first element added is the first one to be removed.
- Queues are commonly used in scenarios where tasks or processes are executed in the order they are received.
- Some common operations associated with queues include:
  - **Enqueue**: Adding an element to the rear (end) of the queue.
  - **Dequeue**: Removing the element from the front (head) of the queue.
  - **Front**: Viewing the element at the front without removing it.
  - **Rear**: Viewing the element at the rear without removing it.
  - **IsEmpty**: Checking if the queue is empty.



Example	Output

```
# Create an empty queue
queue = []

# Add elements onto the queue
queue.append('Chandragupta')
queue.append('Bindusar')
queue.append('Ashoka')

# Display the queue
print("Queue after inserting data:")
print(queue, '\n')

# Display the front of queue
print('Front:\n', queue[0])

# Display the rear of queue
print('Rear:\n', queue[-1])

print("\nElements removed from queue:")
print(" First  -> ",queue.pop(0))
print(" Second -> ",queue.pop(0))
print(" Third  -> ",queue.pop(0))

print("\nQueue after removing elements:")
print(queue)
```

```
Queue after inserting data:
['Chandragupta', 'Bindusar', 'Ashoka']

Front:
 Chandragupta
Rear:
 Ashoka

Elements removed from queue:
 First  ->  Chandragupta
 Second ->  Bindusar
 Third  ->  Ashoka

Queue after removing elements:
[]
```

Looping in Lists

- In python looping through lists can be done in following ways.

Example

```
# Looping in lists using for loop
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Looping in lists Using a for Loop with range()
fruits = ["apple", "banana", "cherry"]
for i in range(len(fruits)):
    print(fruits[i])

# Looping in lists Using a while Loop
fruits = ["apple", "banana", "cherry"]
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1

# Looping in list comprehension
fruits = ["apple", "banana", "cherry"]
[print(fruit) for fruit in fruits]
```

Output

```
apple
banana
cherry

apple
banana
cherry

apple
banana
cherry

apple
banana
cherry
```

The Enumerate Function in Lists

- In Python, the `enumerate()` function is a built-in function that adds a counter to an iterable and returns it as an enumerate object.
- This can be particularly useful when you need both the item and its index while looping over an iterable.

Example

```
colors = ["red", "green", "blue"]
for index, color in enumerate(colors):
    print(index, '--->',color)

# Using a custom start index
for index, color in enumerate(colors, start=1):
    print(index, '--->', color)
```

Output

```
0 ---> red
1 ---> green
2 ---> blue

1 ---> red
2 ---> green
3 ---> blue
```

List Comprehensions

- List comprehensions in Python provide a concise way to create lists. They are often more readable and faster than traditional for loops.
- A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.
- **Syntax**

```
newList = [expression(element) for element in oldList if condition]
```

- **Parameter:**
  - `expression`: Represents the operation you want to execute on every item within the `iterable`.
  - `element`: The term "variable" refers to each value taken from the `iterable`.
  - `iterable`: specify the sequence of elements you want to iterate through. (e.g., a list, tuple, or string).
  - `condition`: (Optional) A filter helps decide whether an element should be added to the new list.
  - `Return`: The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.

Example 1

```
# Finding squares of a list of numbers
numbers = [1, 2, 3, 4, 5]
squared_normal = []

# Normal method
for i in numbers:
    squared_normal.append(i**2)

print('Number List: \n', numbers)
print('List of squares (normal loop):')
print('',squared_normal)

squared_LC = [x ** 2 for x in numbers]
print('List of squares (List Comprehension):')
print('',squared_LC)
```

Result

```
Number List:
 [1, 2, 3, 4, 5]
List of squares (normal loop):
 [1, 4, 9, 16, 25]
List of squares (List Comprehension):
 [1, 4, 9, 16, 25]
```

Example 2

```
# filtering even numbers from a list
# Normal method
even_nums = []
for i in range(1, 10):
    if i%2==0:
        even_nums.append(i)

print('List of even no.s (normal loop):')
print('',even_nums)

# List comprehension method
even_numbers = [num for num in range(1, 10) if num % 2 == 0 ]
print('List of even no.s (List Comprehension):')
print('',even_numbers)
```

Result

```
List of even no.s (normal loop):
 [2, 4, 6, 8]
List of even no.s (List Comprehension):
 [2, 4, 6, 8]
```

Example 3	Result

```
# find vowel in the string "Python"
word = "Python"
vowels = "aeiouAEIOU"

# Normal method
res = []
for i in word:
    if i in vowels:
        res.append(i)
print('List of vowels (normal loop):')
print('',res)

# List comprehension method
result = [char for char in word if char in vowels]
print('List of vowels (List Comprehension):')
print('',result)
```

```
List of vowels (normal loop):
 ['o']
List of vowels (List Comprehension):
 ['o']
```

Functional Programming in Lists

- Python provides several built-in functions based on expressions, which work faster than loop-based user defined code. They are
  - `map()`
  - `reduce()`
  - `filter()`
- **Map Function**
  - The function `map` is used for transforming every value in each sequence by applying a function to it.
  - It takes two input arguments:
    - The **iterable object** (i.e. object which can be iterated upon) to be processed and
    - the **function** to be applied
  - It returns the map object obtained by applying the function to the list.
  - **Syntax**

    ```
    result = map(function, iterable object)
    ```
  - The function to be applied may have been defined already, or it may be defined using a lambda expression which returns a function object.

Example	Output

```
# using the map function
print('Using function:')

def cubes(x):
    return x**3

numlist = [4,-5,2,6,3]
cubelist = list(map(cubes, numlist))
print('List of numbers: \n', numlist)
print('List of cubes: \n', cubelist)


# using the map function: lambda function
print('Using lambda function:')

lamdaCubes = lambda x:x**3
lambdacubelist = list(map(lamdaCubes, numlist))

print('List of numbers: \n', numlist)
print('List of cubes: \n', lambdacubelist)
```

```
Using function:
List of numbers:
 [4, -5, 2, 6, 3]
List of cubes:
 [64, -125, 8, 216, 27]

Using lambda function:
List of numbers:
 [4, -5, 2, 6, 3]
List of cubes:
 [64, -125, 8, 216, 27]
```

- We can even pass more than one sequence in the `map()` function. There are two requirements explained as follows.

- The function must have as many arguments as there are sequences.
- Each argument is called with the corresponding item from each sequence (or none if one sequence is shorter than another).

Example

```
numlist1 = [4,-5,2,6,3]
numlist2 = [3,7,2,-8,2]
print('List 1: \n', numlist1)
print('List 2: \n', numlist2)

print('Sum of Cubes Using Function:')
def sum2cubes(x,y):
    return x**3 + y**3

numlist3 = list(map(sum2cubes, numlist1, numlist2))
print(numlist3)

print('Sum of Cubes Using Lambda Function:')
lamdaCubes = lambda x,y:x**3 + y**3
numlist4 = list(map(sum2cubes, numlist1, numlist2))

print(numlist4)
```

Output

```
List 1:
 [4, -5, 2, 6, 3]
List 2:
 [3, 7, 2, -8, 2]
Sum of Cubes Using Function:
[91, 218, 16, -296, 35]
Sum of Cubes Using Lambda Function:
[91, 218, 16, -296, 35]
```

The `filter()` Function

- The `filter()` function constructs a list from those elements of the list for which a function returns `True`.
- **Syntax**

```
filter(function, sequence)
```

- As per the syntax, the `filter()` function returns a sequence that contains items from the sequence for which the function is `True`.
- Essentially, `filter()` can be used to extract elements from an iterable that meets a certain condition.
- If sequence is a string, Unicode, or a tuple, then the result will be of the same type; otherwise, it is always a list.

Example

```
# using the filter() function
# to check whether numbers are divisible
# by 2 and 3
def check(x):
    if x%2==0 and x%3==0:
        return 1

numList = list(range(1,31))
print('List of numbers: \n', numList)

# using for loop
res23 = []
for i in numList:
    if check(i):
        res23.append(i)
print('Using for loop')
print('List of numbers divisible by 2 and 3: \n',res23)

# using filter()
print('Using filter()')
nums23 = list(filter(check, numList))
print('List of numbers divisible by 2 and 3: \n',nums23)

# using lambda function and filter()
print('using lambda function and filter()')
check23 = lambda x: x%2==0 and x%3==0
nums23new = list(filter(check23, numList))
print('List of numbers divisible by 2 and 3: \n',nums23new)
```

Output

```
List of numbers:
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30]

Using for loop
List of numbers divisible by 2 and 3:
 [6, 12, 18, 24, 30]

Using filter()
List of numbers divisible by 2 and 3:
 [6, 12, 18, 24, 30]

Using lambda function and filter()
List of numbers divisible by 2 and 3:
 [6, 12, 18, 24, 30]
```

The `reduce()` Function

- The `reduce()` function with syntax as given below returns a single value generated by calling the function on the first two items of the sequence, then on the result and the next item and so on
- **Syntax**

```
reduce(function, sequence)
```

- **Key points to remember**
  - If there is only one item in the sequence, then its value is returned.
  - If the sequence is empty, an exception is raised.
  - Creating a list in a very extensive range will generate a `MemoryError` or `OverflowError`.

Example

```
numbers = [6,3,4,7,2,9,1]
sum_result = reduce(lambda x, y: x + y, numbers)
max_result = reduce(lambda a, b: a if a>b else b, numbers)
min_result = reduce(lambda a, b: a if a<b else b, numbers)


print('List of Numbers: \n', numbers)
print('Sum of numbers =',sum_result)
print('Max of numbers =',max_result)
print('Min of numbers =',min_result)
```

Output

```
List of Numbers:
 [6, 3, 4, 7, 2, 9, 1]
Sum of numbers = 32
Max of numbers = 9
Min of numbers = 1
```

- **Optional Initializer Argument**
  - Reduce also accepts an optional third argument, initializer, which is used as the initial value to start the execution.
  - **Syntax**

```
reduce(function, sequence, initializer)
```

  - If the iterable is empty, reduce will return the initializer.
  - Without an initializer, an empty iterable would cause reduce to raise a `TypeError`.

Example	Output

```
from functools import reduce

# Define a function to sum the squares of two numbers
def sum_squares(x, y):
    return x + y**2

# Define a function to sum the cubes of two numbers
sum_cubes = lambda x,y:x+y**3

# List of numbers
numbers = [6,-3,4,-7,2,9,1]

# Use reduce to apply the sum_squares function to the list
result_square = reduce(sum_squares, numbers, 0)
result_cubes = reduce(sum_cubes, numbers, 0)

print('List of Numbers: \n', numbers)
print('Sum of square of numbers = ',result_square)
print('Sum of cube of numbers = ',result_cubes)
```

```
List of Numbers:
 [6, -3, 4, -7, 2, 9, 1]
Sum of square of numbers =  196
Sum of cube of numbers =  648
```

Tuple

- Tuple is a data structure supported by Python used to store collections of data.
- Tuples can store homogeneous as well as heterogeneous data.
- It is very similar to lists but differs in two things.
  - First, a tuple is a sequence of immutable objects. This means that while you can change the value of one or more items in a list, you cannot change the values in a tuple.
  - Second, tuples use parentheses to define its elements whereas lists use square brackets.

Creating and Using Tuples

To create a tuple, you can simply enclose the items within parentheses. For example:

```
mytuple = ("apple", "banana", "cherry")
print(mytuple)

mytuple = tuple(["apple", "banana", "cherry"])
print(mytuple)
```

If you need to create a tuple with only one item, you must include a comma after the item, otherwise, it will not be recognized as a tuple:

```
# Correct way to create a single item tuple
single_item_tuple = ("apple",)

# Incorrect, this is not a tuple but a string
not_a_tuple = ("apple")
```

Tuples can contain items of any data type and can even contain a mix of different types:

```
tuple1 = ("abc", 34, True, 40, "male")
```

Tuple Characteristics

- **Ordered**: The items in a tuple have a specific order that will not change.
- **Unchangeable**: Once a tuple is created, you cannot change, add, or remove items.
- **Allow Duplicates**: Tuples can have items with the same value, which means they can contain duplicates.
- **Indexing**: Each item in a tuple has an index, starting from 0.
- **Lightweight**: They consume relatively small amounts of memory compared to other sequences like lists.
- **Heterogeneous**: They can store objects of different data types and domains, including mutable objects.
- **Nestable**: They can contain other tuples, so you can have tuples of tuples.
- **Iterable**: They support iteration, so you can traverse them using a loop or comprehension while you perform operations with each of their elements.
- **Sliceable**: They support slicing operations, meaning that you can extract a series of elements from a tuple.
- **Combinable**: They support concatenation operations, so you can combine two or more tuples using the concatenation operators, which creates a new tuple.
- **Hashable**: They can work as keys in dictionaries when all the tuple items are immutable.

Accessing Tuple Items

- You can access tuple items by referring to the index number:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1]) # Outputs "banana"
```

- Negative indexing means starting from the end of the tuple:

```
print(thistuple[-1]) # Outputs "cherry"
```

- Like other Python sequences, tuples allow you to extract a portion or slice of their content with a slicing operation, which uses the following syntax:

```
tuple_object[start:stop:step]
```

Example:

```
>>> days = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
            "Saturday", "Sunday",)
>>> print(days[:5])
   ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
>>> days[5:]
   ('Saturday', 'Sunday')
```

Immutability of Tuples

Tuples are immutable, which means that you cannot change tuple items after the tuple has been created. Attempting to do so will result in a TypeError:

```
thistuple = ("apple", "banana", "cherry")
thistuple[1] = "strawberry"      # Raises a TypeError
del thistuple[1]                 # Raises a TypeError
```

Tuple Length

To determine the number of items in a tuple, use the `len()` function:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple)) # Outputs 3
```

Tuple Operations

Tuples support various operations like concatenation, nesting, repetition, and slicing. For instance, you can concatenate tuples using the + operator:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(tuple1 + tuple2) # Outputs (1, 2, 3, 4, 5, 6)
```

Converting Between Tuples and Other Types

You can convert a tuple to a list to modify it, and then convert it back to a tuple:

```
mylist = list(mytuple)
mylist.append("orange")
mytuple = tuple(mylist)
```

Tuples can store any type of object, including mutable ones. This means that you can store lists, sets, dictionaries, and other mutable objects in a tuple. You can change the content of mutable objects even if they're nested in a tuple.

```
student_info = ("Linda", 18, ["Math", "Physics", "History"])
print(student_info[2][2])          # prints "History"

student_info[2][2] = "Computer Science"
print(student_info[2][2])          # prints "Computer Science"
```

Packing and Unpacking of Tuples

- Python has the notion of packing and unpacking tuples.
- For example, when you write an assignment statement like `point = x, y, z`, you're ***packing*** the values of `x`, `y`, and `z` in point. That's how you create new tuple objects.
- You can also do the inverse operation and ***unpack*** the values of a tuple into an appropriate number of variables.

```
>>> point = (7, 14, 21)          # Packing
>>> x, y, z = point              # Unpacking
>>> x
    7
>>> y
    14
>>> z
    21
```

- One can use the * operator in tuples pack and unpack the elements in tuples.

Example	Output

`Tup1 = (1,3,4,2,6,7,5)`
`*val1, val2 = Tup1`
`print('val1:\n',val1)`
`print('val2:\n',val2)`
 	
`val3, *val4 = Tup1`
`print('val3:\n',val3)`
`print('val4:\n',val4)`
 	
`val5, *val6, val7 = Tup1`
`print('val5:\n',val5)`
`print('val6:\n',val6)`
`print('val7:\n',val7)`
	`val1:`
	` [1, 3, 4, 2, 6, 7]`
	`val2:`
	` 5`
	`val3:`
	` 1`
	`val4:`
	` [3, 4, 2, 6, 7, 5]`
	`val5:`
	` 1`
	`val6:`
	` [3, 4, 2, 6, 7]`
	`val7:`
	` 5`

Example	Output

`Tup1 = (1,3,4,2)`
`Tup2 = (4,3,5,3)`
`print((Tup1, Tup2))`
`print((Tup1 + Tup2))`
`print((*Tup1, *Tup2))`
	`((1, 3, 4, 2), (4, 3, 5, 3))`
	`(1, 3, 4, 2, 4, 3, 5, 3)`
	`(1, 3, 4, 2, 4, 3, 5, 3)`

Basic Tuple Operations

- Like strings and lists, you can also perform operations like concatenation. repetition, etc on tuples.
- The only difference is that a new tuple should be created when a change is required in an existing tuple.

Operation	Expression	Output
Length	`len((4,5,3,5,2,3,7,9))`	`8`
Concatenation	`(1,2,3) + (4,5,6)`	`(1,2,3,4,5,6)`
Repetition	`("good.",)*3`	`('good.','good.','good.')`
Membership	`5 in (4,3,5,2,7,9)`	`True`
Iteration	`for i in (4,3,5,2,7,9):` `  print(i, end='')`	`4,3,5,2,7,9`

Comparison
(use <,>,==)
	`T1 = (1,2,3,4,5,6)`
	`T1 = (1,2,3,4,5,6)`
	`print(T1>T2)`
		`False`

Maximum	`max((4,3,5,2,7,9))`	`9`
Minimum	`min((4,3,5,2,7,9))`	`2`
Convert into a tuple (converts a sequence into tuple)	`tuple("hello")`	`('h', 'e', 'l', 'l', 'o')`

	`tuple([4,3,5,2,7,9])`	`(4,3,5,2,7,9)`

Index: returns the index of the first occurrence of the element `Tup.index(element)`	`T1 = (1,2,3,4,5,6)`
	`print(T1.index(3))`
	`print(T1.index(8))`
		
		`2`
		`ValueError`

Index: Find within a range defined between start and stop `Tup.index(element,start,stop)`	`T1 = (4,3,5,2,7,9)` `print(T1.index(7,2,5))`	`4`

Count: returns the number of elements with a specific value	`T1=`
	`(1,2,3,4,5,6,5,3,5,6)`
	`print(T1.count(3))`
	`print(T1.count(25))`
		
		`2`
		`0`

Sum (not defined for strings)	`T1 = (4,3,5,2,7,9)` `print(T1.sum())`	`30`

The `zip` function

The function `zip` is used to produces a zip object (iterable object), whose i^{th} element is a tuple containing i^{th} element from each iterable object passed as argument to the zip function.

Example

```
# working of the zip function
colors = ('red', 'yellow', 'orange')
fruits = ['cherry', 'banana','orange']
quantity = ('1 kg', 12, '2 kg')
fruitColor = list(zip(colors, fruits))
print(fruitColor)
print()
fruitColorQuantity1 = list(zip(fruits, colors, quantity))
print(fruitColorQuantity1)
print()
fruitColorQuantity2 = list(zip(fruitColor, quantity))
print(fruitColorQuantity2)
print()
```

Output

```
[('red', 'cherry'), ('yellow', '
banana'), ('orange', 'orange')]

[('cherry', 'red', '1 kg'), ('ba
nana', 'yellow', 12), ('orange',
'orange', '2 kg')]

[(('red', 'cherry'), '1 kg'), ((
'yellow', 'banana'), 12), (('ora
nge', 'orange'), '2 kg')]
```

List Comprehension and Tuples

- The list comprehension concept can be extended to tuples to manipulate the values of one tuple to create a new tuple.

Example

```
# list comprehension in tuples
def double(T):
    return ([i*2 for i in T])
Tup = (1,2,3,4,5)
print('Normal method:')
print("Original values: ", Tup)
print("Double values  : ", double(Tup))
print()

print('Using comprehension:')
doubles1 = tuple(x*2 for x in Tup)
doubles2 = list(x*2 for x in Tup)
print('Creating a tuple:',doubles1)
print('Creating a list:',doubles2)
```

Output

```
Normal method:
Original values:  (1, 2, 3, 4, 5)
Double values  :  [2, 4, 6, 8, 10]

Using comprehension:
Creating a tuple: (2, 4, 6, 8, 10)
Creating a list: [2, 4, 6, 8, 10]
```

Dictionary

- Dictionary is a data structure in which we store values as a pair of key and value. Each key is separated from its value by a colon (:), and consecutive items are separated by commas. The entire items in a dictionary are enclosed in curly brackets({}).
- The syntax for defining a dictionary is

```
dictionary_name = {key_1:value_1, key_2:value_2, key_3:value_3}
```

or

```
dictionary_name = {key_1:value_1,
                   key_2:value_2,
                   key_3:value_3}
```

- The keys in the dictionary must be **unique** and be of any immutable data type (like strings, numbers, or tuples).
- There is no stringent requirement for uniqueness and type of values. That is, value of a key can be of any type.
- Dictionaries are not sequences, rather they are mappings. Mappings are collections of objects that store objects by key instead of by relative position.
- Dictionary keys are case-sensitive. Two keys with the same name but in different case are not the same in Python.

Creating Dictionaries

Example

```
print('Creating an empty dictionary:')
Dict1 = {}
print(Dict1)

print('Creating a dictionary with values:')
Dict2 = {'Name' : 'Arav',
         'Course' : 'BTech',
         'Branch' : 'CSE',
         'Specialization' : 'AIML',
         'Roll_No' : '16/001'}
print(Dict2)

print('Creating a dictionary with comprehension:')
Dict3 = {x: 2**x for x in range(1, 11)}
print(Dict3)
```

Output

```
Creating an empty dictionary:
{}
Creating a dictionary with values:
{'Name': 'Arav', 'Course': 'BTech', 'Branch': 'CS
E', 'Specialization': 'AIML', 'Roll_No': '16/001'
}
Creating a dictionary with comprehension:
{1: 2, 2: 4, 3: 8, 4: 16, 5: 32, 6: 64, 7: 128, 8
: 256, 9: 512, 10: 1024}
```

Accessing, Adding and Modifying Values in Dictionaries

Example

```
Dict = {'Name' : 'Arav',
         'Course' : 'BTech',
         'Branch' : 'CSE',
         'Spec' : 'AIML',
         'Roll_No' : '16/001'}

# Accessing the values of dictionaries
print('Name          :', Dict['Name'])
print('Course        :', Dict['Course'])
print('Branch        :', Dict['Branch'])
print('Specialization:', Dict['Spec'])
print('Roll No       :', Dict['Roll_No'])

# Adding a new key:value pair
Dict['CGPA'] = 9.67
print('CGPA          :', Dict['CGPA'])

# Changing a key:value pair
Dict['Name'] = 'Arav Gupta'
print('Name          :', Dict['Name'])
```

Output

```
Name          : Arav
Course        : BTech
Branch        : CSE
Specialization: AIML
Roll No       : 16/001

CGPA          : 9.67

Name          : Arav Gupta
```

Deletion of Dictionary Items

- One or more items can be deleted by using the `del` command.
- To remove all items `clear()` function can be used.
- To entirely delete the dictionary from the memory `del` command can be used.

Example

```
Dict = {'Name' : 'Arav',
        'Course' : 'BTech',
        'Branch' : 'CSE',
        'Spec' : 'AIML',
        'Roll_No' : '16/001'}
print('Original:')
print(Dict)

del Dict['Course']
print('\nAfter deletion of one item:')
print(Dict)

Dict.clear()
print('\nAfter clearing:')
print(Dict)

del Dict
print('\nAfter deletion of the whole:')
print(Dict)
```

Output

```
Original:
{'Name': 'Arav', 'Course': 'BTech', 'Branch': 'CS
E', 'Spec': 'AIML', 'Roll_No': '16/001'}

After deletion of one item:
{'Name': 'Arav', 'Branch': 'CSE', 'Spec': 'AIML',
'Roll_No': '16/001'}

After clearing:
{}

After deletion of the whole:
Traceback (most recent call last):
  File "C:\Users\gyana\OneDrive\CM_FDP\AIML\MLW C
ourse 2024\Programs\dictDemo.py", line 62, in <mo
dule>
    print(Dict)
NameError: name 'Dict' is not defined. Did you me
an: 'Dict1'?
```

Beware of duplicate keys

- Keys must have unique values. Not even a single key can be duplicated in a dictionary.
- If you try to add a duplicate key, then the last assignment is retained. This is shown in the example given below.

Example

```
Dict = {'Name' : 'Arav',
        'Course' : 'BTech',
        'Branch' : 'CSE',
        'Spec' : 'AIML',
        'Roll_No' : '16/001',
        'Name': 'Manas'}
print('Effect of having duplicate keys:')
print('Name          :', Dict['Name'])
print('Course        :', Dict['Course'])
print('Branch        :', Dict['Branch'])
print('Specialization:', Dict['Spec'])
print('Roll No       :', Dict['Roll_No'])
```

Output

```
Effect of having duplicate keys:
Name          : Manas
Course        : BTech
Branch        : CSE
Specialization: AIML
Roll No       : 16/001
```

- In a dictionary, keys should be strictly of a type that is immutable.
- This means that a key can be of strings, number, or tuple type but it cannot be a list which is mutable.
- In case you try to make your key of a mutable type, then a `TypeError` will be generated as shown below.

Example

```
79 # Tuples and lists as keys of dictionary
80 d1 = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
81 print('d1:\n',d1)
82
83 d2 = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
84 print('d1:\n',d2)
```

Output

```
d1:
 {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
Traceback (most recent call last):
  File "C:\Users\gyana\OneDrive\CM_FDP\AIML\MLW Course 2024\Pr
ograms\dictDemo.py", line 83, in <module>
    d2 = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
TypeError: unhashable type: 'list'
```

Built-in Dictionary Functions and Methods

Operation	Description	Example	Output
`len(Dict)`	Returns the length of dictionary, the number of items (key-value pairs)	`Dict1 = {'Roll_No':'16/001' 'Name': 'Arav', 'Course': 'BTech'}` `print(len(Dict1))`	`3`
`str(Dict)`	Returns a string representation of the dictionary	`Dict1 = {'Roll_No':'16/001' 'Name': 'Arav', 'Course': 'BTech'}` `print(str(Dict1))`	`{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech'}`
`Dict.clear()`	Removes all the elements from the dictionary		`{}`
`Dict.copy()`	Returns a shallow copy of the dictionary		

`dict.fromkeys(seq, val)`	Returns a dictionary with the specified keys and value	`Subjects = [ 'ICP', 'DS', 'AA', 'ToC']`
		`Score = -1`
		`Marks = dict.fromkeys (Subjects, Score)`
		`print(Marks)`
			`{'ICP': -1, 'DS': -1, 'AA': -1, 'ToC': -1}`

`Dict.get(key)`	Returns the value of the specified key	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.get('Name'))`	`Arav`
`Dict.items()`	Returns a list containing a tuple for each key value pair	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.items())`	`dict_items([('Roll_No', '16/001'), ('Name', 'Arav'), ('Course', 'BTech')])`
`Dict.keys()`	Returns a list containing the dictionary's keys	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.keys())`	`dict_keys(['Roll_No', 'Name', 'Course'])`
`Dict.pop()`	Removes the element with the specified key		
`Dict.popitem()`	Removes the last inserted key-value pair		

`Dict.setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
		`Dict1.setdefault('Marks', 0)`
		`print(Dict1)`
			`{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech', 'Marks': 0}`

`Dict.update()`	Updates the dictionary with the specified key-value pairs	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
		`Dict2 = {'Marks': 86, 'Grade':'A'}`
		`Dict1.update(Dict2)`
		`print(Dict1)`
			`{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech', 'Marks': 86, 'Grade': 'A'}`

`Dict.values()`	Returns a list of all the values in the dictionary	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.values())`	`dict_values(['16/001', 'Arav', 'BTech'])`

`in / not in`	Checks whether a given key is present in the dictionary or not.	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
		`print('Name' in Dict1)`
		`print('Marks' in Dict1)`
			`True`
			`False`

Looping in Dictionaries

Example	Output

`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
`for i, j in Dict1.items():`
`  print(i, '=',j)`
	`Roll_No = 16/001`
	`Name = Arav`
	`Course = BTech`

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed.
  - Set items are unchangeable, but you can remove items and add new items.
- Sets are written with curly brackets.
- Sets are unordered, so you cannot be sure in which order the items will appear.
- Set items are unordered, unchangeable, and do not allow duplicate values.
  - **Unordered**
    - Unordered means that the items in a set do not have a defined order.
    - Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
  - **Unchangeable**
    - Set items are unchangeable, meaning that we cannot change the items after the set has been created.
    - Once a set is created, you cannot change its items, but you can remove items and add new items
  - **Duplicates Not Allowed**
    - Sets cannot have two items with the same value.
    - **Example**

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

Output

```
{'apple', 'cherry', 'banana'}
```

The `set()` Constructor

- It is also possible to use the `set()` constructor to make a set.
- **Examples**

```
thisset1 = set(("apple", "banana", "cherry")) # set using a tuple
thisset2 = set(["apple", "banana", "cherry"]) # set using a list
thisset3 = set({"apple":1, "banana":2, "cherry":3})
                                        # set using a dictionary

print(thisset1)
print(thisset2)
print(thisset3)
```

Output

```
{'banana', 'apple', 'cherry'}
{'banana', 'apple', 'cherry'}
{'banana', 'apple', 'cherry'}
```

Set Items - Data Types

- Set items can be of any data type, both homogeneous and heterogeneous.
- **Examples**

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
set1 = {"abc", 34, True, 40, "male"}
```

Access Set Items

- You cannot access items in a set by referring to an index or a key.
- But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.
- **Example**
  - Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
      print(x)
```

Output

```
banana
apple
cherry
```

- **Example**
  - Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
print("orange" in thisset)
```

Output

```
True
False
```

Set Methods and Functions

Operation	Description
`len(setName)`	Determines how many items a set has
`setName.add(item)`	Adds an element to the set
`setName.clear()`	Removes all the elements from the set
`setName.copy()`	Returns a copy of the set
`set1.difference(set2)`	Returns a set containing the difference between two or more sets
`set1.difference_update(set2)`	Removes the items in this set that are also included in another, specified set
`setName.discard(item)`	Remove the specified item
`set1.intersection(set2)`	Returns a set, that is the intersection of two other sets
`set1.intersection_update(set2)`	Removes the items in this set that are not present in other, specified set(s)
`set1.isdisjoint(set2)`	Returns whether two sets have a intersection or not
`set1.issubset(set2)`	Returns whether another set contains this set or not
`set1.issuperset(set2)`	Returns whether this set contains another set or not
`setName.pop()`	Removes an element from the set
`setName.remove(item)`	Removes the specified element
`set1.symmetric_difference(set2)`	Returns a set with the symmetric differences of two sets

`set1.symmetric_difference_update(set2)`	inserts the symmetric differences from this set and another
`set1.union(set2)`	Return a set containing the union of sets
`set1.update(set2)`	Update the set with the union of this set and others
`set1 == set2 or set1 != set2`	Returns True (or False) if sets are equivalent

Change Items

- Once a set is created, you cannot change its items, but you can add new items.
- **Add Items**
  - Once a set is created, you cannot change its items, but you can add new items.
  - To add one item to a set use the `add()` method.
  - **Example**
    - Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Output

```
{'banana', 'apple', 'orange', 'cherry'}
```

- **Add Sets, Tuple, List or Dictionary**
  - To add items from another set, tuple, list or dictionary into the current set, use the `update()` method.
  - **Example**
    - Add elements from `tropical` into `thisset`:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
                                # Works with tuple, list or dictionary
thisset.update(tropical)
print(thisset)
```

Output

```
{'pineapple', 'mango', 'banana', 'papaya', 'apple', 'cherry'}
```

Remove Set Items

- To remove an item in a set, use the `remove()`, or the `discard()` method.
- If the item to remove does not exist, `remove()` will raise an error, `discard()` will NOT raise an error.
- **Example**
  - Remove `"banana"` by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Output

```
{'apple', 'cherry'}
```

- **Example**
  - Remove `"banana"` by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
```

```
print(thisset)
```

Output

```
{'apple', 'cherry'}
```

- You can also use the `pop()` method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.
- The `clear()` method empties the set.
- The `del` keyword will delete the set completely.

Join Two Sets

- The `union()` method returns a **new** set containing all items from both sets.
  - **Example**

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

Output

```
{1, 'a', 2, 3, 'b', 'c'}
```

- The `update()` inserts the items in `set2` into `set1`.
  - **Example**

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)       # z = set1|set2 also works
print(set3)
```

Output

```
{1, 'a', 2, 3, 'b', 'c'}
```

- Both `union()` and `update()` will exclude any duplicate items.

Keep ONLY the Duplicates

- The `intersection()` method will return a **new** set, that only contains the items that are present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)   # z = x&y also works
print(z)
```

Output

```
{'apple'}
```

- The `intersection_update()` method will keep only the items that are present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)
```

Output

```
{'apple'}
```

Keep All, But NOT the Duplicates

- The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.
  - **Example**

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

  Output

```
{'google', 'cherry', 'microsoft', 'banana'}
```

- The `symmetric_difference_update()` method will return a new set, that contains only the elements that are NOT present in both sets.
  - **Example**

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

  Output

```
{'google', 'cherry', 'microsoft', 'banana'}
```

Keep Difference Only

- The `difference()` method will return a new set, that contains only the elements that are present in one set but not in other.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z1 = x.difference(y)
print(z1)
z2 = x.difference(y)
print(z2)
```

  Output

```
{'banana', 'cherry'}
{'microsoft', 'google'}
```

- The `difference_update()` method will return a new set, that contains only the elements that are NOT present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.difference_update(y)
print(x)
```

  Output

```
{'banana', 'cherry'}
```

Checking if Subset

  - **Example**

```
x = {1,2,3,4,5,6}
y = {1,2,3,4,5,6,7,8,9,10}
# Checking if subset
print(x.issubset(y))
```

```
print(x<=y)
```

Output

```
True
True
```

Checking if Superset

- **Example**

```
x = {1,2,3,4,5,6}
y = {1,2,3,4,5,6,7,8,9,10}
# Checking if superset
print(y.issuperset(x))
print(y>=x)
```

Output

```
True
True
```

Checking if Disjoint

- **Example**

```
x1 = {1,2,3,4,5,6}
y1 = {1,2,3,4,5,6,7,8,9,10}
# Checking if disjoint
print(y1.isdisjoint(x1))

x2 = {1,2,3}
y2 = {4,5,6,7,8,9,10}
# Checking if disjoint
print(y2.isdisjoint(x2))
```

Output

```
False
True
```

Checking if Equivalent or not

- **Example**

```
x1 = {1,2,3,4,5,6}
y1 = {1,2,3,4,5,6}
# Checking if equivalent
print(x1==y1)
print(x1!=y1)

x2 = {1,2,3}
y2 = {4,5,6,7,8,9,10}
# Checking if equivalent
print(y2==x2)
print(x2!=y2)
```

Output

```
True
False
False
True
```

Difference between tuple, list, set, dictionary

Parameters	List	Tuple	Set	Dictionary
Definition	A list is an ordered, mutable collection of elements.	A tuple is an ordered, immutable collection of elements.	A set is an unordered collection of unique elements.	A dictionary is an unordered collection of key-value pairs.
Syntax	Syntax includes square brackets [,] with ',' separated data.	Syntax includes curved brackets (,) with ',' separated data.	Syntax includes curly brackets {,} with ',' separated data.	Syntax includes curly brackets {,} with ',' separated key-value data.
Creation	A list can be created using the list() function or simple assignment to [].	Tuple can be created using the tuple() function.	A set dictionary can be created using the set() function.	A dictionary can be created using the dict() function.
Empty Data Structure	An empty list can be created by l = [].	An empty tuple can be created by t = ().	An empty set can be created by s = set().	An empty dictionary can be created by {}.
Order	It is an ordered collection of data.	It is also an ordered collection of data.	It is an unordered collection of data.	Ordered collection in Python version 3.7, unordered in Python Version=3.6.
Duplicate Data	Duplicate data entry is allowed in a List.	Duplicate data entry is allowed in a Tuple.	All elements are unique in a Set.	Keys are unique, but two different keys CAN have the same value.
Indexing	Has integer based indexing that starts from '0'.	Also has integer based indexing that starts from '0'.	Does NOT have an index based mechanism.	Has a Key based indexing i.e. keys identify the value.
Addition	New items can be added using the append() method.	Being immutable, new data cannot be added to it.	The add() method adds an element to a set.	update() method updates specific key-value pair.
Deletion	pop() method allows deleting an element.	Being immutable, no data can be popped/deleted.	Elements can be randomly deleted using pop().	pop(key) removes specified key along with its value.
Sorting	sort() method sorts the elements.	Immutable, so sorting method is not applicable.	Unordered, so sorting is not advised.	Keys are sorted by using the sorted() method.
Search	index() returns index of first occurrence.	index() returns index of first occurrence.	Unordered, so searching is not applicable.	get(key) returns value against specified key.
Reversing	reverse() method reverses the list.	Immutable, so reverse method is not applicable.	Unordered, so reverse is not advised.	No integer-based indexing, so no reversal.
Count	count() method returns occurrence count.	count() method returns occurrence count.	count() not defined for sets.	count() not defined for dictionaries.

Recursion

- A recursive function is a function that calls itself, where successive calls reduce a computation to smaller computations of the same type until a base case with a trivial solution is reached.

  - **Example**

```
def power(r, n):
## iterative definition of power function
value = 1
for i in range(1, n + 1):
      value = r * value
      return value

print(power(2, 3))
```

    Output

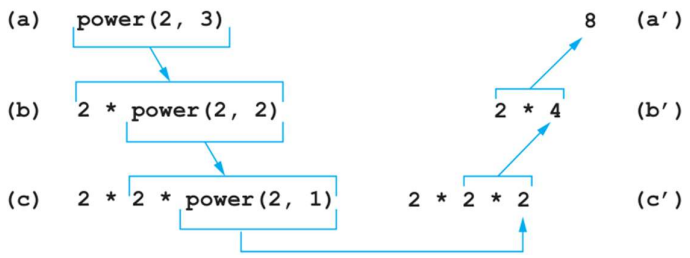
```
8
```

  - **Example**

```
def power(r, n):
## recursive definition of power function
if n == 1:
      return r
else:
      return r * power(r, n - 1)
print(power(2, 3))
```

    Output

```
8
```



- Recursive algorithms have two traits.
  - There are one or more **base cases** with trivial solutions.
  - There is an "**inductive step**" that successively reduces the problem to smaller versions of the same problem, with the reduction eventually culminating in a base case. This inductive step is called the reducing step.
- **Pseudocode of recursion**

```
if a base case is reached
      Solve the base case directly.
else
      Repeatedly reduce the problem to a version increasingly
      closer to a base case until it becomes a base case.
```

- **Advantages of Recursion**
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.
  - We can reduce the length of code and become more readable and understandable to the user/ programmer.
- **Disadvantages of Recursion**
  - Sometimes the logic behind recursion is hard to follow through.
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
  - Recursive functions are hard to debug.
  - Recursion can lead to stack overflow errors if the recursion depth is too high.

Error and Exception Handling

- Python may stop execution of a program because of two reasons.
  - **Syntax Errors**
    - An **error** is an action that is incorrect or inaccurate.
      - This is caused by wrong syntax in the code. It leads to the termination of the program.
  - **Exceptions**
    - Exceptions are raised when some internal events occur which change the normal flow of the program.
    - Some of the exceptions are listed below.

Exception	Description
`ArithmeticError`	Raised when an error occurs in numeric calculations
`AssertionError`	Raised when an assert statement fails
`AttributeError`	Raised when attribute reference or assignment fails
`EOFError`	Raised when the `input()` method hits an "end of file" condition (EOF)
`FloatingPointError`	Raised when a floating point calculation fails
`GeneratorExit`	Raised when a generator is closed (with the `close()` method)
`ImportError`	Raised when an imported module does not exist
`IndentationError`	Raised when indentation is not correct
`IndexError`	Raised when an index of a sequence does not exist
`KeyError`	Raised when a key does not exist in a dictionary
`KeyboardInterrupt`	Raised when the user presses Ctrl+c, Ctrl+z or Delete
`LookupError`	Raised when errors raised cant be found
`MemoryError`	Raised when a program runs out of memory
`NameError`	Raised when a variable does not exist
`NotImplementedError`	Raised when an abstract method requires an inherited class to override the method
`OSError`	Raised when a system related operation causes an error
`OverflowError`	Raised when the result of a numeric calculation is too large
`ReferenceError`	Raised when a weak reference object does not exist
`RuntimeError`	Raised when an error occurs that do not belong to any specific exceptions
`StopIteration`	Raised when the `next()` method of an iterator has no further values
`SyntaxError`	Raised when a syntax error occurs
`TabError`	Raised when indentation consists of tabs or spaces
`SystemError`	Raised when a system error occurs
`SystemExit`	Raised when the `sys.exit()` function is called
`TypeError`	Raised when two different types are combined
`UnboundLocalError`	Raised when a local variable is referenced before assignment
`UnicodeError`	Raised when a unicode problem occurs
`UnicodeEncodeError`	Raised when a unicode encoding problem occurs
`UnicodeDecodeError`	Raised when a unicode decoding problem occurs
`UnicodeTranslateError`	Raised when a unicode translation problem occurs
`ValueError`	Raised when there is a wrong value in a specified data type
`ZeroDivisionError`	Raised when the second operator in a division is zero

- It's important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to **gracefully handle** errors and prevent the program from crashing.

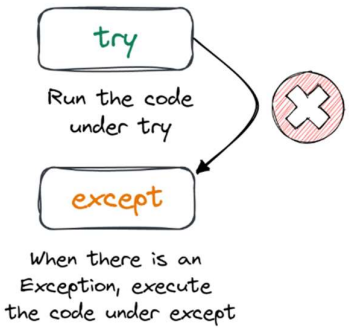
Why use Exception

- **Standardized error handling**: Using built-in exceptions or creating a custom exception with a more precise name and description, you can adequately define the error event, which helps you debug the error event.
- **Cleaner code**: Exceptions separate the error-handling code from regular code, which helps us to maintain large code easily.
- **Robust application**: With the help of exceptions, we can develop a solid application, which can handle error event efficiently
- **Exceptions propagation**: By default, the exception propagates the call stack if you don't catch it. For example, if any error event occurred in a nested function, you do not have to explicitly catch-and-forward it; automatically, it gets forwarded to the calling function where you can handle it.
- **Different error types**: Either you can use built-in exception or create your custom exception and group them by their generalized parent class, or Differentiate errors by their actual class

Exception Handling using try ..... catch

- Since exceptions abnormally terminate the execution of a program, it is important to handle exceptions. In Python, we use the `try...except` block to handle exceptions.

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```



- Here, we have placed the code that might generate an exception inside the `try` block. Every try block is followed by an `except` block.
- When an exception occurs, it is caught by the `except` block. The `except` block cannot be used without the `try` block.

Example: Without Exception Handling

```
1    numerator = 10
2    denominator = 0
3
4    result = numerator/denominator
5
6    print(result)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-8f43cf287b9c> in <module>
      2 denominator = 0
      3
----> 4 result = numerator/denominator
      5
      6 print(result)

ZeroDivisionError: division by zero
```

Example: Exception Handling using try.....catch

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")
```

```
Error: Denominator cannot be 0.
```

- For each `try` block, there can be zero or more `except` blocks. Multiple `except` blocks allow us to handle each exception differently.
- The argument type of each `except` block indicates the type of exception that can be handled by it.

Example: Handling multiple exceptions using try.....catch

```
try:
    a = int(input("Enter value of a:"))
    b = int(input("Enter value of b:"))
    c = a/b
    print("The answer of a divide by b:", c)
except ValueError:
    print("Entered value is wrong")
except ZeroDivisionError:
    print("Can't divide by zero")
```

```
Enter value of a:44
Enter value of b:0
Can't divide by zero
```

```
Enter value of a:44
Enter value of b:abc
Entered value is wrong
```

```
Enter value of a:44
Enter value of b:45
The answer of a divide by b: 0.9777777777777777
```

Handle multiple exceptions with a single except clause

- We can also handle multiple exceptions with a single except clause.
- For that, we can use a `tuple` of values to specify multiple exceptions in an `except` clause.

Example: Handling multiple exceptions using a single except

```
try:
    a = int(input("Enter value of a:"))
    b = int(input("Enter value of b:"))
    c = a / b
    print("The answer of a divide by b:", c)
except(ValueError, ZeroDivisionError):
    print("Please enter a valid value")
```

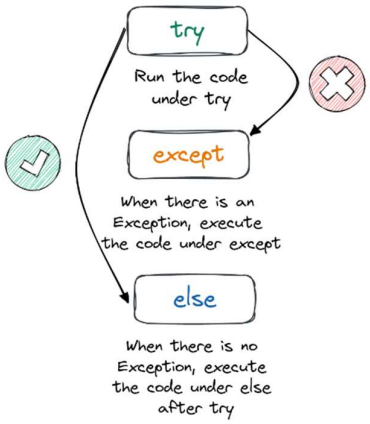
```
Enter value of a:44
Enter value of b:0
Please enter a valid value
```

```
Enter value of a:44
Enter value of b:abc
Please enter a valid value
```

```
Enter value of a:44
Enter value of b:45
The answer of a divide by b: 0.9777777777777777
```

Exception handling with try.....catch.....else

- In some situations, we might want to run a certain block of code if the code block inside `try` runs without any errors.
- For these cases, you can use the optional `else` keyword with the `try` statement.



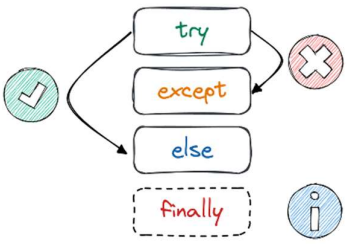
Example: Handling multiple exceptions using try.....catch.....else

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

```
Enter a number: 45
Not an even number!

Enter a number: 44
0.022727272727272728
```

Finally Keyword in Python

- The `finally` keyword in the `try-except` block is always executed, irrespective of whether there is an exception or not.
- In simple words, the `finally` block of code is run after the `try`, except, the `else` block is final. It is quite useful in cleaning up resources and closing the object, especially closing the files.



- The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

Example: Handling multiple exceptions using try.....catch.....else.....finally

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
finally:
    print("This is finally block.")
```

```
Enter a number: 44
0.022727272727272728
This is finally block.

Enter a number: 45
Not an even number!
This is finally block.
```

Raising an Exceptions

- In Python, the `raise` statement allows us to throw an exception. The single arguments in the `raise` statement show an exception to be raised. This can be either an exception object or an Exception class that is derived from the Exception class.

- The `raise` statement is useful in situations where we need to raise an exception to the caller program. We can raise exceptions in cases such as wrong data received or any validation failure.
- Follow the below steps to raise an exception:
  - Create an exception of the appropriate type. Use the existing built-in exceptions or create your own exception as per the requirement.
  - Pass the appropriate data while raising an exception.
  - Execute a raise statement, by providing the exception class.

Example: Raising an exception

```
1  def simple_interest(amount, year, rate):
2      try:
3          if rate > 100:
4              raise ValueError(rate)
5          interest = (amount * year * rate) / 100
6          print('The Simple Interest is', interest)
7          return interest
8      except ValueError:
9          print('interest rate is out of range', rate)
10
11 print('Case 1')
12 simple_interest(800, 6, 8)
13 print()
14
15 print('Case 2')
16 simple_interest(800, 6, 800)
```

```
Case 1
The Simple Interest is 384.0

Case 2
interest rate is out of range 800
```

File Handling

- File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations.
- Python file handling refers to the process of working with files on the filesystem. It involves operations such as reading from files, writing to files, appending data, and managing file pointers.
- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

- The following operations can be performed with files.
  - Opening
  - Reading
  - Writing
  - Adding (Appending)
  - Closing

Types of File

- **Text File**: Text file usually we use to store character data. For example, test.txt
- **Binary File**: The binary files are used to store binary data such as images, video files, audio files, etc.

File Path

A file path defines the location of a file or folder in the computer system. There are two ways to specify a file path.

1. **Absolute path**: which always begins with the root folder
2. **Relative path**: which is relative to the program's current working directory

Different Modes to Open a File in Python

Mode	Description
`r`	It opens an existing file to read-only mode. The file pointer exists at the beginning.
`rb`	It opens the file to read-only in binary format. The file pointer exists at the beginning.
`r+`	It opens the file to read and write both. The file pointer exists at the beginning.
`rb+`	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
`w`	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name.
`wb`	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists.

`w+`	It opens the file to write and read data. It will override existing data.
`wb+`	It opens the file to write and read both in binary format
`a`	It opens the file in the append mode. It will not override existing data. It creates a new file if no file exists with the same name.
`ab`	It opens the file in the append mode in binary format.
`a+`	It opens a file to append and read both.
`ab+`	It opens a file to append and read both in binary format.

Opening Files

- In Python, we need to open a file first to perform any operations on it—we use the `open()` function to do so.
- Let's look at an example:
  - Suppose we have a file named `file1.txt`.
  - To open this file, we can use the `open()` function.
    ```
    file1 = open("file1.txt")
    ```
  - Here, we have created a file object named file1. Now, we can use this object to work with files.

Example: Opening file with relative path

```
# Opening the file in the same folder
fp = open('FH.txt', 'r')
# read file
for each in fp:
    print(each)
# Closing the file after reading
```

```
Welcome to MLW01

This is a sample.txt

Line 3

Line 4

Line 5
```

Example: Opening file with absolute path

```
1  # Opening the file with absolute path
2  fp = open(r"C:\Users\gyana\OneDrive\CM_FDP\AIML\MLW Course 2024\Programs\FH.txt", 'r')
3
4  # read file
5  print(fp.read())
6  # Closing the file after reading
7  fp.close()
8
9  # Reading a part of the file
10 fp = open('FH.txt', "r")
11 print (fp.read(7))
12 fp.close()
```

```
Welcome to MLW01
This is a sample.txt
Line 3
Line 4
Line 5
Welcome
```

- We can also split lines while reading files in Python. The `split()` function splits the variable when space is encountered. You can also split using any characters as you wish.

Example

```python
# Python code to illustrate split() function
with open("FH.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

```
['Welcome', 'to', 'MLW01']
['This', 'is', 'a', 'sample.txt']
['Line', '3']
['Line', '4']
['Line', '5']
```

Writing to a File

- To write content into a file, Use the access mode `w` to open a file in a write mode.
- **Note:**
  - If a file already exists, it truncates the existing content and places the file handle at the beginning of the file. A new file is created if the mentioned file doesn't exist.
  - If you want to add content at the end of the file, use the access mode a to open a file in append mode

Example: writing to a file

```python
text = "This is new content"

# writing new content to the file
fp = open('FH.txt', 'w')
fp.write(text)
print('Done Writing')
fp.close()

# reading the file contents
fp = open('FH.txt', 'r')
for each in fp:
    print(each)

# Closing the file after reading
fp.close()
```

```
Done Writing
This is new content
```

- We can also use the written statement along with the `with()` function.

Example: writing to a file using with()

```python
# Python code to illustrate with() alongwith write()
with open("FH.txt", "w") as fp:
    fp.write("Writing line 1\n")
    fp.write("Writing line 2")

fp = open('FH.txt', 'r')
for each in fp:
    print(each)

# Closing the file after reading
fp.close()

```

```
Writing line 1

Writing line 2
```

Append Mode

- Let us see how the append mode works.

Example

```python
# Python code to illustrate append() mode
file = open('FH.txt', 'a')
file.write("\nThis will add this line")
file.close()

fp = open("FH.txt", 'r')
# read file
print(fp.read())
# Closing the file after reading
fp.close()
```

```
Writing line 1
Writing line 2
This will add this line
```

Move File Pointer

- The `seek()` method is used to change or move the file's handle position to the specified location.
- The cursor defines where the data must be read or written in the file.
- The position (index) of the first character in files is zero, just like the string index.

Example

```python
fp = open("FH.txt", "r")
# move to 11 character
fp.seek(11)
# read from 11th character
print(fp.read())
# Closing the file after reading
fp.close()
```

```
MLW01
This is a sample.txt
Line 3
Line 4
Line 5
```

- The `tell()` method to return the current position of the file pointer from the beginning of the file.

Example

```python
f = open("FH.txt", "r")
# read first line
f.readline()
# get current position of file handle
print(f.tell())
```

```
18
```

Copy Files

- There are several ways to cop files in Python.
- The `shutil.copy()` method is used to copy the source file's content to the destination file.

Example

```python
import shutil

src_path = "FH.txt"
dst_path = "FH_copy.txt"
shutil.copy(src_path, dst_path)
print('Copied')
```

```
Copied
```

File Object Methods

Method	Description
`read()`	Returns the file content.
`readline()`	Read single line
`readlines()`	Read file into a list
`truncate(size)`	Resizes the file to a specified size.
`write()`	Writes the specified string to the file.
`writelines()`	Writes a list of strings to the file.
`close()`	Closes the opened file.
`seek()`	Set file pointer position in a file
`tell()`	Returns the current file location.
`fileno()`	Returns a number that represents the stream, from the operating system's perspective.
`flush()`	Flushes the internal buffer.

Rename Files

- In Python, the `os` module provides the functions for file processing operations such as renaming, deleting the file, etc.
- The `os` module enables interaction with the operating system.
- The `os` module provides `rename()` method to rename the specified file name to the new name. The syntax of `rename()` method is shown below.

Example

```
# File Rename

import os

# Absolute path of a file
old_name = "FH.txt"
new_name = "FH_Rename.txt"

# Renaming the file
os.rename(old_name, new_name)
```

Functions to delete files and folders

Function	Description
`os.remove('file_path')`	Removes the specified file.
`os.unlink('file_path')`	Removes the specified file. Useful in UNIX environment.
`pathlib.Path("file_path").unlink()`	Delete the file or symbolic link in the mentioned path
`os.rmdir('empty_dir_path')`	Removes the empty folder.
`pathlib.Path(empty_dir_path).rmdir()`	Unlink and delete the empty folder.
`shutil.rmtree('dir_path')`	Delete a directory and the files contained in it.

- **Note:**
  - All above functions delete files and folders permanently.
  - The `pathlib` module was added in Python 3.4. It is appropriate when your application runs on different operating systems.

Delete Files

- In Python, the `os` module provides the `remove()` function to remove or delete file path.

Example

```
import os

# remove file with absolute path
os.remove("FH_copy.txt")
```

Check if File exist:

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

Example

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

Delete Folder

- To delete an entire folder, use the `os.rmdir()` method.
- You can only remove ***empty*** folders.

Example

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

Implementing all the functions in File Handling

- In this example, we will cover all the concepts that we have seen above.
- Other than those, we will also see how we can delete a file using the `remove()` function from Python `os` module .

Example

```
import os

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

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

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

def rename_file(filename, new_filename):
```

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

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


if __name__ == '__main__':
    filename = "example.txt"
    new_filename = "new_example.txt"

    create_file(filename)
    read_file(filename)
    append_file(filename, "This is some additional text.\n")
    read_file(filename)
    rename_file(filename, new_filename)
    read_file(new_filename)
    delete_file(new_filename)
```

Result

```
File example.txt created successfully.
Hello, world!

Text appended to file example.txt successfully.
Hello, world!
This is some additional text.

File example.txt renamed to new_example.txt successfully.
Hello, world!
This is some additional text.

File new_example.txt deleted successfully.
```

Advantages of File Handling in Python

- **Versatility**: File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.
- **Flexibility**: File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files , etc.), and to perform different operations on files (e.g. read, write, append, etc.).
- **User – friendly**: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.
- **Cross-platform**: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling in Python

- **Error-prone:** File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks**: File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity**: File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.
- **Performance**: File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.