RANKA
DATE / /
PAGE

Week-4 Advanced Python

Lec-1 File handling + Serialisation & Deserialisation

File handling -

Types of data used for I/O :

Text - '12345' sequence of unicod chars (text file)

Binary - 12345 sequence of bytes (binary file)

How File I/O is done in programming language

- open a file
- Read/write data
- close the file

Text file

1) # If file is not present.

```
f = open('sample.txt', 'w')
f.write('Hello world')
f.close()
```

(f → file handler)

2) Write Multiline string

```
f = open('sample1.txt', 'w')
f.write('hello world')
f.write('\n how are you?')
f.close()
```

3) If the file is already present.

```
f = open('sample.txt', 'w')
f.write('salman khan')
f.close()
```

ऊपर हम पुरानी file में कुछ लिखते हैं Our old text get vanished.

how open works

how exactly append works?

```
f = open('sample.txt', 'a')
f.write('\n I am fine')
f.close()
```

Write multiple lines

```
L = ['hello\n', 'hi\n', 'how are you\n', 'I am fine']
f = open('sample.txt', 'w')
f.writelines(L)
f.close()
```

Reading from file.

```
f = open('sample.txt', 'r')
s = f.read()
print(s)
f.close()
```

Reading upto n chars

```
f = open('sample.txt', 'r')
s = f.read(10)
print(s)
f.close()
```

To read line by line

```
f = open('sample.txt', 'r')
print(f.readline())
Print(f.read line())
f.close()
```

Print (f.readline(), end='')

Readline → help to read a very large file [take data in parts]
Read → when data is less

```
f = open('sample text','r')
while True:
    data = f.readline()
    if data == "":
        break
    else:
        print(data, ends='')
f.close().
```

Using ~~with~~ Context Manager (With)

```
with open('sample.txt','w') as f:
    f.write('Salman bhai')
```

अगर with keyword use करते हैं file अपने आप close हो जाती है

~~try f.read now.~~

```
with open('sample.txt','r') as f
    print(f.read())
```

moving within a file → 10 char

```
with open('sample.txt','r') as f:
    print(f.read(10))
    print(f.read(10))
```

```
with open('big.txt','r') as f
    chunk_size = 10
    while (len(f.read(chunk_size)) > 0 :
        print (f.read(chunk_size), end = '')
        f.read(chunk_size)
```

seek & tell function

```
with open('sample.txt','r') as f:
    print (f.read(10))
    print (f.tell())
    f.seek(0)
    print (f.read(3))
```

tell → आगे cursor की position बताता है

seek → हमें जहाँ cursor चाहिए वहाँ ले जाता है

seek during write

```
with open('sample.txt','w') as f:
    f.write('Hello')
    f.seek(0)
    f.write('ka')
```

Working with binary file:

```
with open('screenshot1.jpg','rb') as f:
    with open('screenshot-copy.jpg','wb') as wf:
        wf.write(f.read())
```

Working with other data types:

```
with open('sample.txt','w') as f:
    f.write(5)
```

Note: we can store file it usually accept utf ; str. not other data type

RANKA
DATE / /
PAGE

Serialisation and Deserialisation

Serialisation : process of converting python data type to JSON format.
Deserialisation : process of converting JSON to python data type

JSON – Java script Object Notation (Universal text format)

```
# serialisation using JSON module.
# list
L = [1, 2, 3, 4]
with open ('demo.json', 'w') as f
    json.dump (L, f)

# dictionary
d = {
    'name' : 'nitish'
    'age' : 33
    'gender' : 'male'
}
with open ('demo.json', 'w') as f
    json.dump (d, ff, indent = 4)

# deserialisation
import json
with open ('demo.json', 'r') as f:
    d = json.load (f)
print (d)
print (type (d))
```

RANKA
DATE / /
PAGE

Note: Tuple datatype will be store like a list either serialise or desserialise

Serializing and Deserializing Custom object

```
class Person:
    def __init__(self, fname, lname, age, gender):
        self.fname = fname
        self.lname = lname
        self.age = age
        self.gender = gender

person = Person('Nitish', 'Singh', 33, 'male')

import json
def show_object(person):
    if isinstance(person, Person):
        return "{} {} age -> {} gender -> {}".format(person.fname,
            person.lname, person.age, person.gender)

with open('demo.json', 'w') as f
    json.dump(peson, f, default=show_object)
```

read deserialising

```
import json
with open('demo.json', 'r') as f:
print(json.load(f))
```

RANKA
DATE / /
PAGE

Pickling

pickling: It is the process whereby a Python object hierarchy is converted into a byte stream, and

unpickling: It is the inverse operation, whereby a byte stream (from a binary file or byte-like object) is converted back into object hierarchy.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age,

    def display_info(self):
        print('Hi my name is', self.name, 'and I am', self.age, 'years old')

p = Person('nitish', 33)
```

\# pickle dump

```
import pickle
with open('person.pkl', 'wb') as f:
    pickle.dump(p, f)
```

\# pickle load

```
import pickle
with open('person.pkl', 'rb') as f:
    p = pickle.load(f)
p.display_info()
```

pickle

object retain its power.

RANKA
DATE / /
PAGE

pickle v/s JSON

pickle lets the user to store data in binary format. JSON lets the user store data in human readable text format

multithreading

Week-9

RANKA
DATE / /
PAGE

Exception handling

There are 2 stages where error may happen in a program.

- During compilation → Syntax error
- During Execution → Exception.

Syntax error:

- Something in your programe not written according to program grammar
- Error is raised by interpreter / Compiler.
- you can solve it by rectify the program

Ex:

Index Error:

- The index error is thrown when trying to access an item at an invalid index.

Module Not Found Error

- The error is thrown when module could not be found.

Key Error

- The key error is thrown when key is not found in dictionary.

TypeError:

The error is thrown when operation is applied to an object of inappropriate type.

```
1 + 'a'
```

ValueError:

Stack

RANKA
DATE / /
PAGE

~~NameErr~~

NameError: It is thrown when an object could not be found.

print(A)

Attribute Error: To execute a function on some data type which do not even exist.

Exception: If things go wrong during the execution of program (running)
It happens when something unforseen happen.

Eg: Memory overflow.
Divide by 0 → logical error
Database error

Why is it important to handle exception.
how to handle exception.
Try except block

```
with open('sample.txt', 'w') as f:
    f.write()

try:
    with open('sample.txt','r') as f:
    print(f.read())
except:
    print("sorry file not found")
```

try
except
else
finally

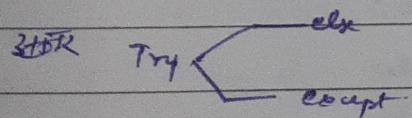
RANKA
DATE / /
PAGE

Catching specific exception

```
try:
    f = open('sample1.txt', 'r')
    print(f.read())
    print(m)
except NameError:
    print("variable not exist")
except Exception as e:
    print(e.with_traceback) or print(e)
```

else: block

```
try:
    f = open('sample.txt', 'r')
except FileNotFoundError:
    print('File nahi mili')
except Exception:
    print('Kuch toh lafda hai')
else:
    print(f.read()
```



finally

```
try:
    f = open('sample.txt', 'r')
except FileNotFound:
    print('file nahi mili')
finally:
    print('yeh toh print hoga hi')
```

Java
try - try
except - catch
raise - throw

RANKA
DATE / /
PAGE

raise Exception :

- In python programming, Exception are raised when error occur at runtime.
- we can manually raise exception using raise keyword.

raise ModuleNotFoundError ('aise hi try kar raha tha')

```
class Bank :
    def __init__(self, balance) :
        self.balance = balance.

    def withdraw (self, amount)
        if amount < 0 :
            raise Exception ('Amount can not be Negative')
        if self.balance < amount :
            raise Exception('paise nahi hai tere pas')
        self.balance = self.balanca - amount

obj = Bank (10000)
try :
    obj.withdraw (15000)
except Exception as e :
    print(e)
else :
    print (obj.balance)
```

Why need to create Custom Class

Full control

RANKA
DATE / /
PAGE

Creating Custom Exception :

Exception hierarchy in python

```
class MyException(Exception):
    def __init__(self, message):
        print(message)
```

↓

```
class Bank:
```

Ex:

```
class SecurityError(Exception):
    def __init__(self, message):
        print(message)
    def logout(self):
        print('logout')

class Google:
    def __init__(self, name, email, password, device):
        self.name = name
        self.password = password
        self.email = email
        self.device = device

    def login(self, email, pass, device):
        if device != self.device:
            raise SecurityError('bhai teri toh lag gayi')
        if email == self.email and password == self.password:
            print('welcome')
        else:
            print('login error')
```

```
obj = Google('nitish', 'nitish@gmail.com', '1234', 'android')
try:
    obj.login('nitish@gmail.com', '1234', 'windows')
except SecurityError as e:
    e.layout()
else:
    print(obj.name)
finally:
    print('database closed')
```

Name space:

A namespace is a space that holds name (identifiers). Programmatically speaking, namespace are dictionary of identifiers (key) & their object (values)

There are 4 types of namespaces:

- Builtin Namespace
- Global Namespace
- Enclosing Namespace
- Local Namespace

Scope:

A scope is a textual region in python where name space is directly accessible.

Scope → box

name space → dictionary of variable inside scope

RANKA
DATE / /
PAGE

LEGB rule:-

local → enclosing → global → built in scope.

Local & global:-

```
a = 2
def temp():
    b = 3
    print(b)
temp()
print(a)
```

Note: We can access variable from Gloabal scope in local scope but we cannot change them.

Or

```
global a
a += 1
print(a)
```

built-in scope:-

```
import builtins
print(dir(builtins))
```

```
L = [1,2,3]
max(L)
```

```
def max():
    print('hello')
max(L)
```

Enclosing scope:

Enclosing scope:
- usually occur in nested function.

```
def outer():
    def inner():
        print("Inner")
    inner()
    print("outer")
outer()
print("main")
```

Decorators

A decorator in python is a function that recieves another function as input and add some functionality (decoration) to and it and returns it.

Note: python function are 1st class citize (can execute all the function/operation)

There are 2 types of decorators available in python.

1) built in decorators like
   @staticmethod, @classmethod, @abstractmethod
2) user defined decorators.

```
def modify(func, num):
    return func(num)

def square(num):
    return num**2

modify(square, 2)
```

#

```
def my_decorator(func)
    def wrapper():
        print("*********")
        func()
        print("*********")
    return wrapper

def hello()
    print('hello')

a = my_decorator(hello)
a()
```

```
def my_decorator(func)
    def wrapper():
        print('---------')
        func()
        print('---------')
    return wrapper

@my_decorator
def hello():
    print('hello')
hello()
```

```
import time
def timer(func):
    def wrapper():
        start = time.time()
        func()
        print('time taken by', func.__name__, time.time() - start, 'sec')
    return wrapper

@timer
def hello():
    print('hello world')
    time.sleep(2)
```

~~What is~~

Iteration:

It is a general term for taking each item of something, one after another. i.e, To go over items, group of items that is iteration.

```
for i in range(len(
```

Iterator:

An iterator is an object that allows the programmer to traverse through a sequence of data without having to store the entire data in the memory.

```
L = [x for x in range(1,100000)]
for i in L:
    print(i*2)
```

```
import sys
sys.getsizeof(L)/64.
```

1) The process of traversing an object is called Iteration

2) Iterator object है जिसकी help से Iteration होता है।

What is Iterable.

Iterable is an object, which one iterate over.

3) An object over which one can iterate called Iterable.

```
L = [1,2,3]
type(L)
type(iter(L))
```

L → Iterable
iter(L) → Iterator

Note: Every Iterator is Iterable
But Not all Iterable are Iterators.

Trick: how to determine whether an object is Iterator or Iterable

Iterable: अगर किसी object पर loop चल जाता है तो वह Iterable नहीं तो नहीं है।

or

```
dir(a)
```

Iterator: अगर कोई Object Iterator है तो We can find it (dir)

dir(a) ⟶ next, iter

```
L = [1, 2, 3]
iter_L = iter(L)
iter_L
```

Understanding how Loop work

```
num = [1, 2, 3]
      * * * * *  ------------> fetch the iterator
iter_num = iter(num)
# step 2 --> next
next(iter_num)
next(iter_num)
next(iter_num)
```

Making our own for loop:

```
def mera_khudka_for_loop(iterable):
    Iterator = iter(iterable)
    while True:
        try:
            print(next(iterator))
        except StopIteration:
            break
```

A Confusing point

```
num = [1,2,3]
iter_obj = iter(num)
print(id(iterobj), 'Address of iterator1')
iter_obj2 = iter(iter_obj)
print(id(iter_obj2), 'Address of iterator2')
```

Generators & python gener

```
class mera-rang:
    def --init-- (self, start, end):
        self.start = start
        self.end = end

    def --iter--(self):
        return mera-iterator(self)


class mera-iterator:
    def --init-- (self, iterable_obj):
        self.iterable = iterable-obj

    def --iter-- (self):
        return self

    def --next-- (self):
        if self.iterable.start >= self.iterable.end:
            raise StopIteration

        current = self.iterable.start
        self.iterable.start += 1
        return current
```

Generators:-

It is a simple way of Iterator.

```
def gen_demo():
    yield " first statement"
    yield " second statement"
    yield " third statement"

gen = gen_demo()
print(gen)
print(next(gen))   → Executing first yield statement
print(next(gen))
```

difference b/w yield & return

Note:- A simple function execute & remove from memory but generator it gradually executing.

Range function using Generator

~~for i in~~

```
def mera-range (start, end):
    for i in range (start, end):
        yield i;

for i in mera-range (15,20)
    print(i)
```

Generator Expression

List - comprehension:

```
L= [i**2 for i in range (1,101)]
gen = (i**2 for i in range (1,101))
for i in gen:
    print(i)
```

Advantages

1) Ease of Implementation
2) Memory efficient
3) Representing Infinite Streams

```
def all-even():
    n=0
    while True:
        yield n
        n+=2
even_num_gen= all_even
```

4) Chaining Generators

```
def fibonacci-numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci-numbers(10))))
```