

# Advance Python Programming Language

---

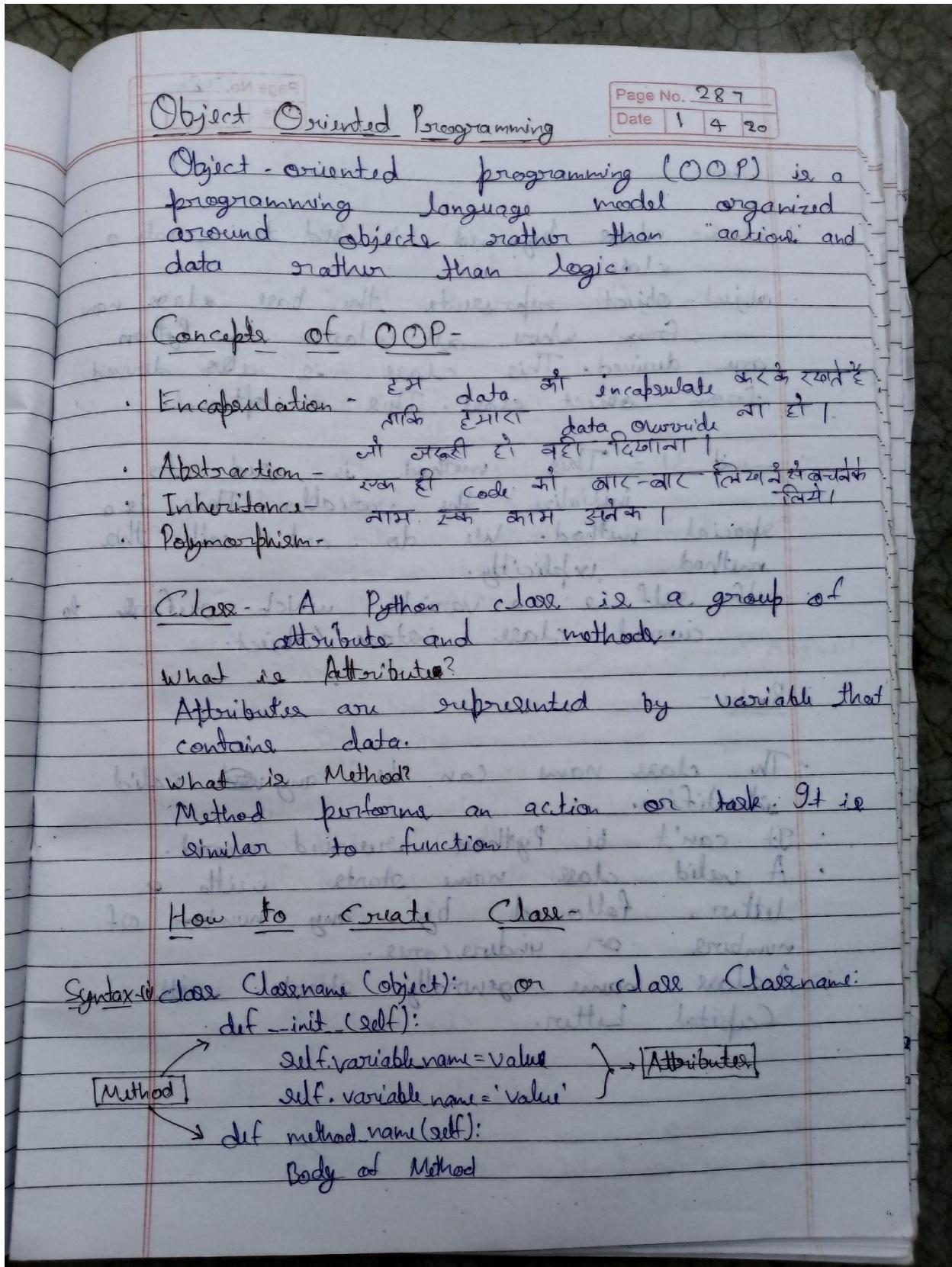
GEEKYSHOWS YOUTUBE CHANNEL LEARNING NOTES

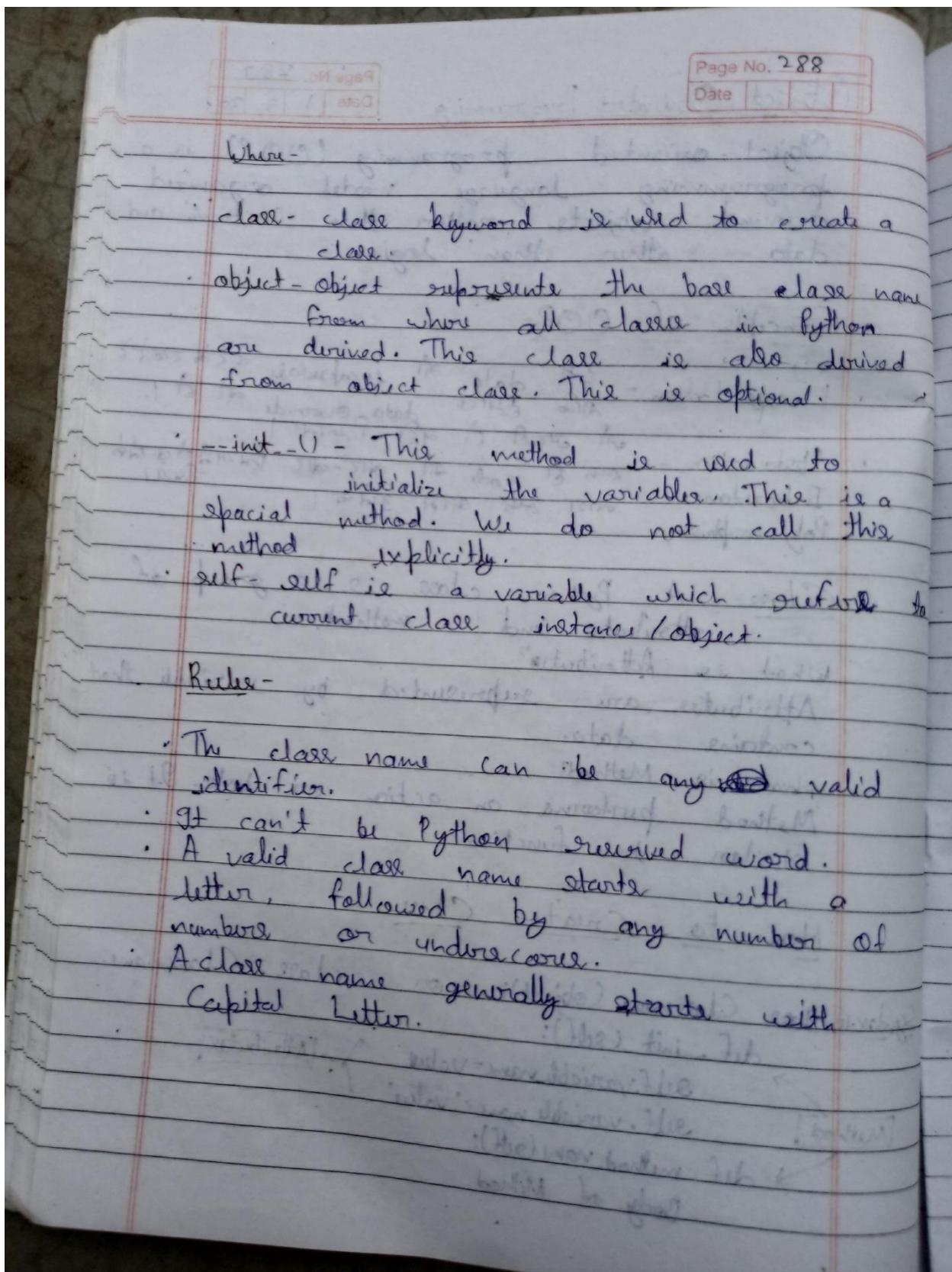
Source Code- [https://github.com/satyam-seth-learnings/python\\_learning/tree/main/Geeky%20Shows/Advance%20Python](https://github.com/satyam-seth-learnings/python_learning/tree/main/Geeky%20Shows/Advance%20Python)

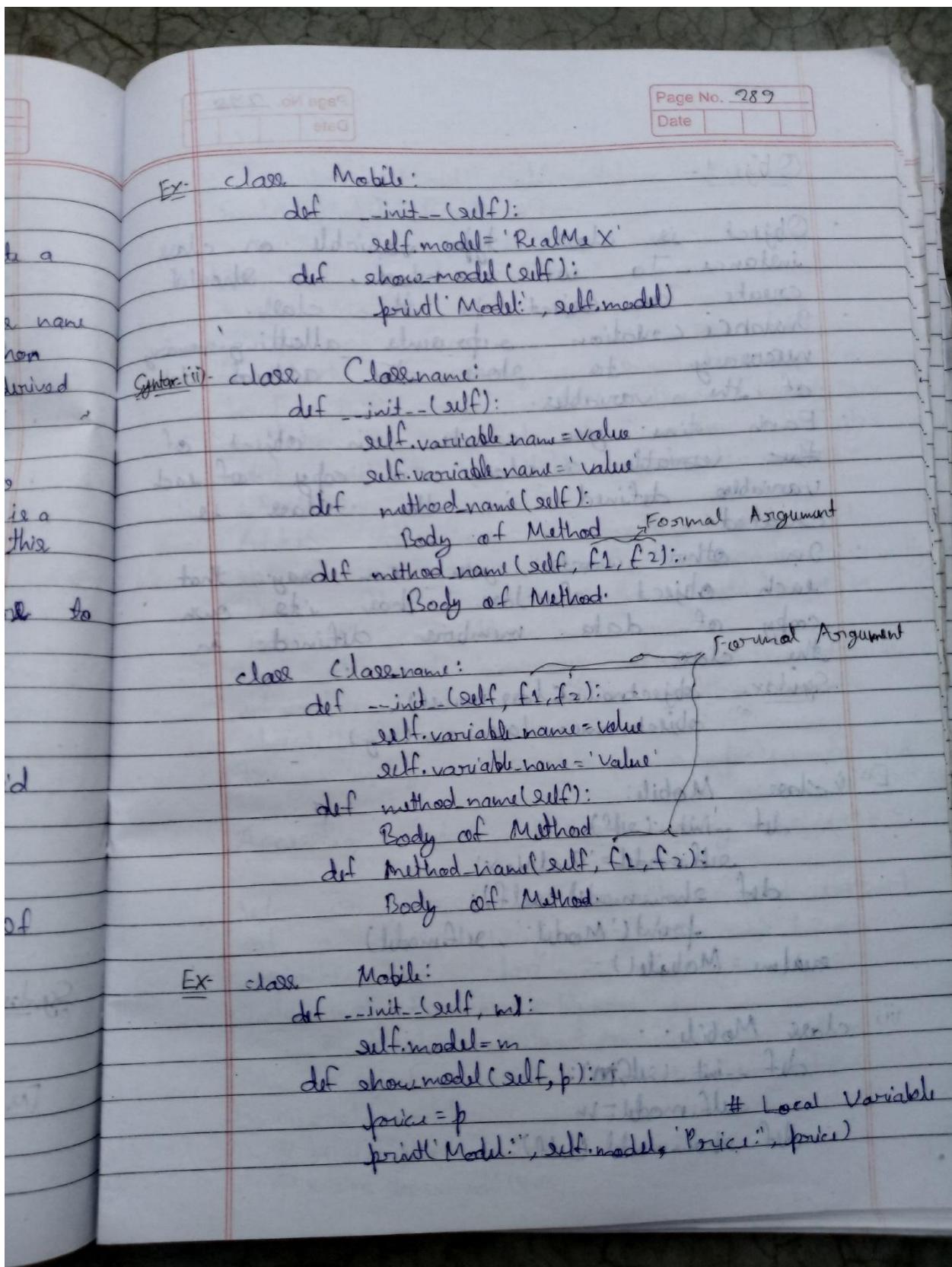
Playlist Link- [https://youtube.com/playlist?list=PLbGui\\_ZYuhijd1hUF2VWiKt8FHNBa7kGb](https://youtube.com/playlist?list=PLbGui_ZYuhijd1hUF2VWiKt8FHNBa7kGb)

SATYAM SETH

21-09-2021







Page No. 290  
Date

Object-

- Object is class type variable or class instance. To use a class, we should create an object to the class.
- Instance creation represents allotting memory necessary to store the actual data of the variables.
- Each time you create an object of the variable a class a copy of each variable defined in the class is created.
- In other words you can say that each object of a class has its own copy of data members defined in the class.

Syntax-

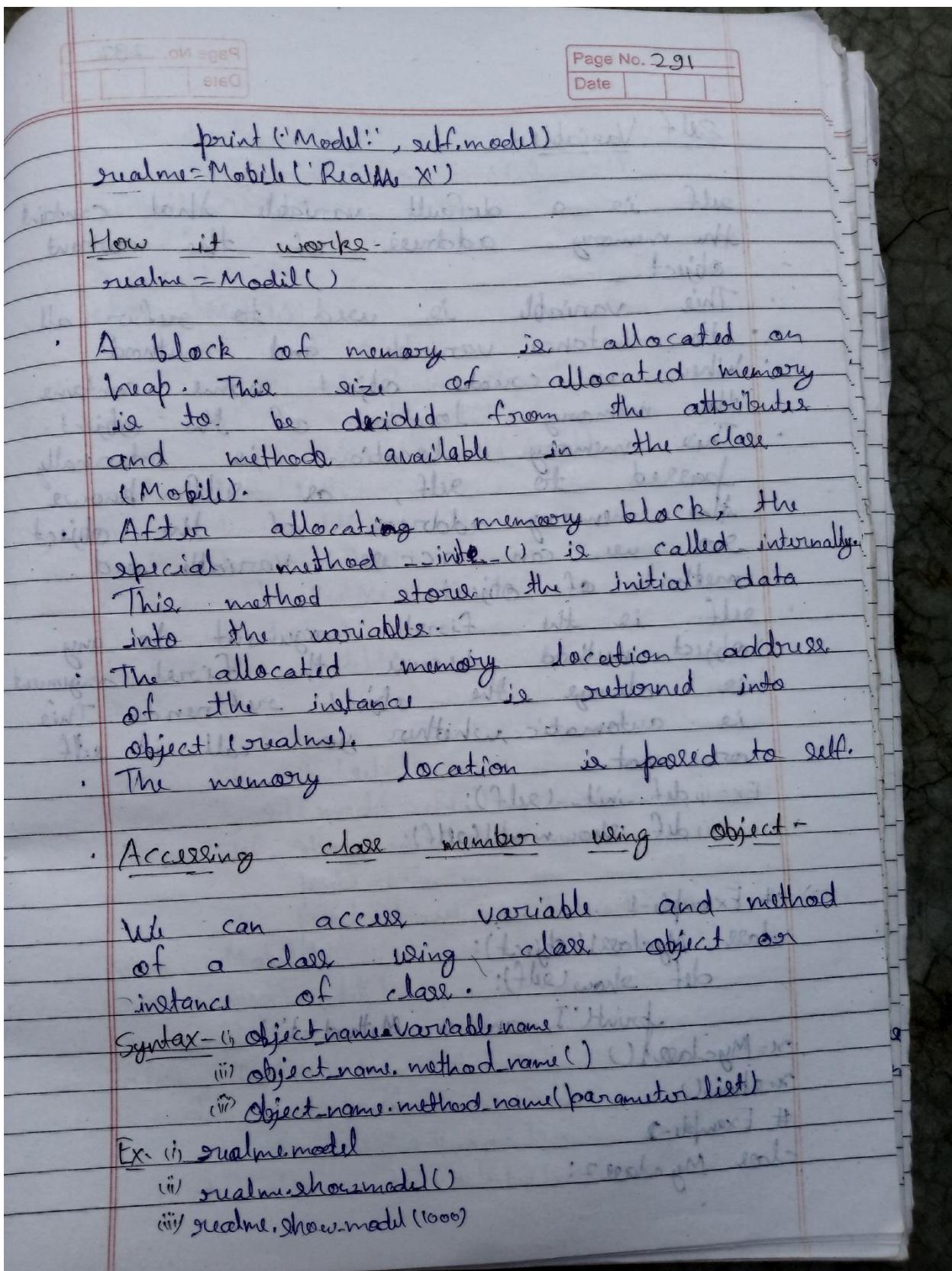
```
objectname=Class_name()
or
objectname=Class_name(arg)
```

Ex (i) class Mobile:

```
def __init__(self):
    self.model='RealMeX'
def showmodel(self):
    print('Model:', self.model)
realme=Mobile()
```

Ex (ii) class Mobile:

```
def __init__(self,m):
    self.model=m
def showmodel(self):
```



Page No. 292

Date

### Self Variable -

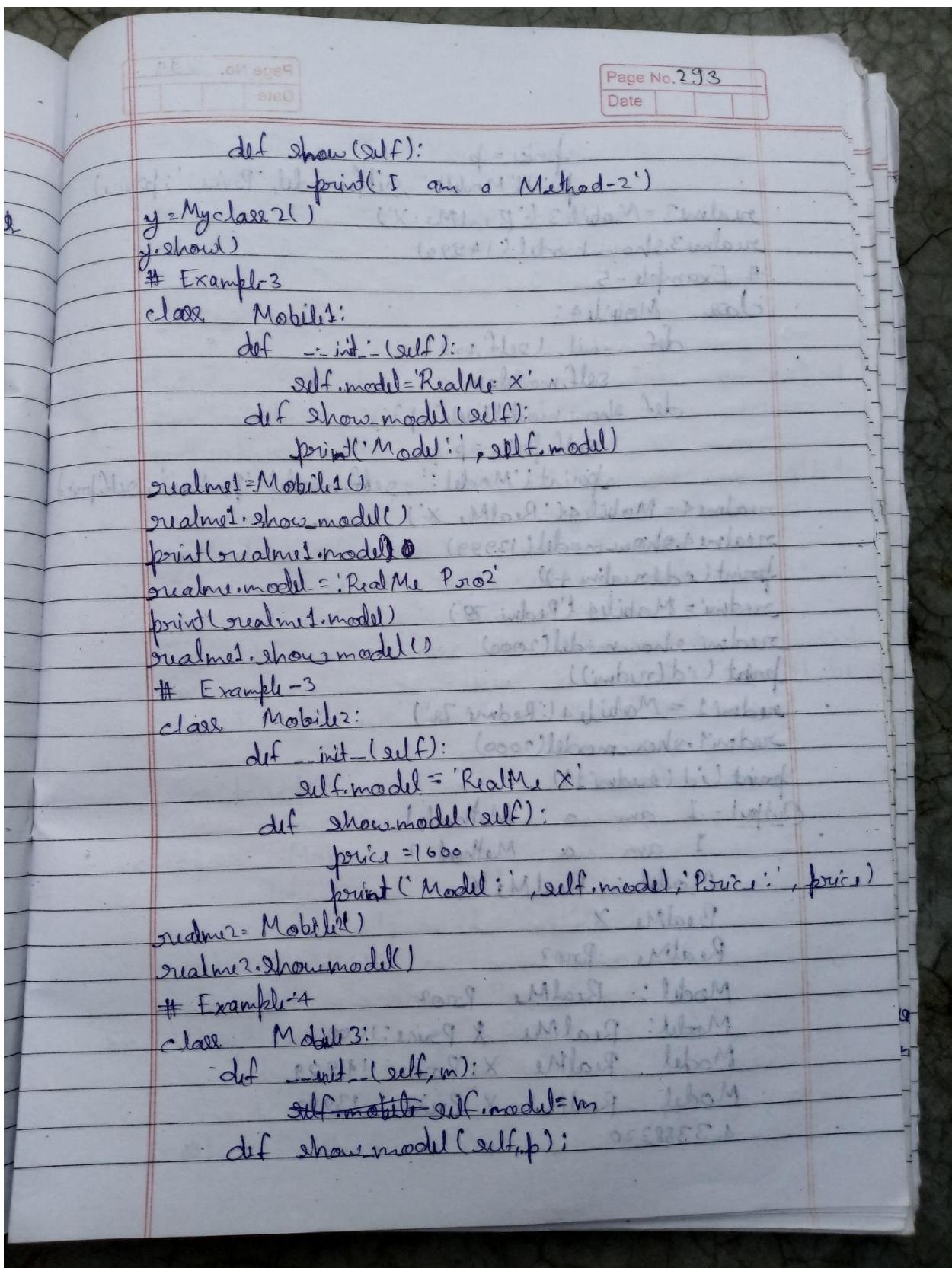
- self is a default variable that contains the memory address of the current object.
- This variable is used to refer all the instance variable and method.
- When we create object name contains the memory location of the object.
- This memory location is internally passed to self, as self knows the memory address of the object so we can access variable and method of object.
- self is the first argument to any object method because the first argument is always the object reference. This is automatic whether you call it self or not.

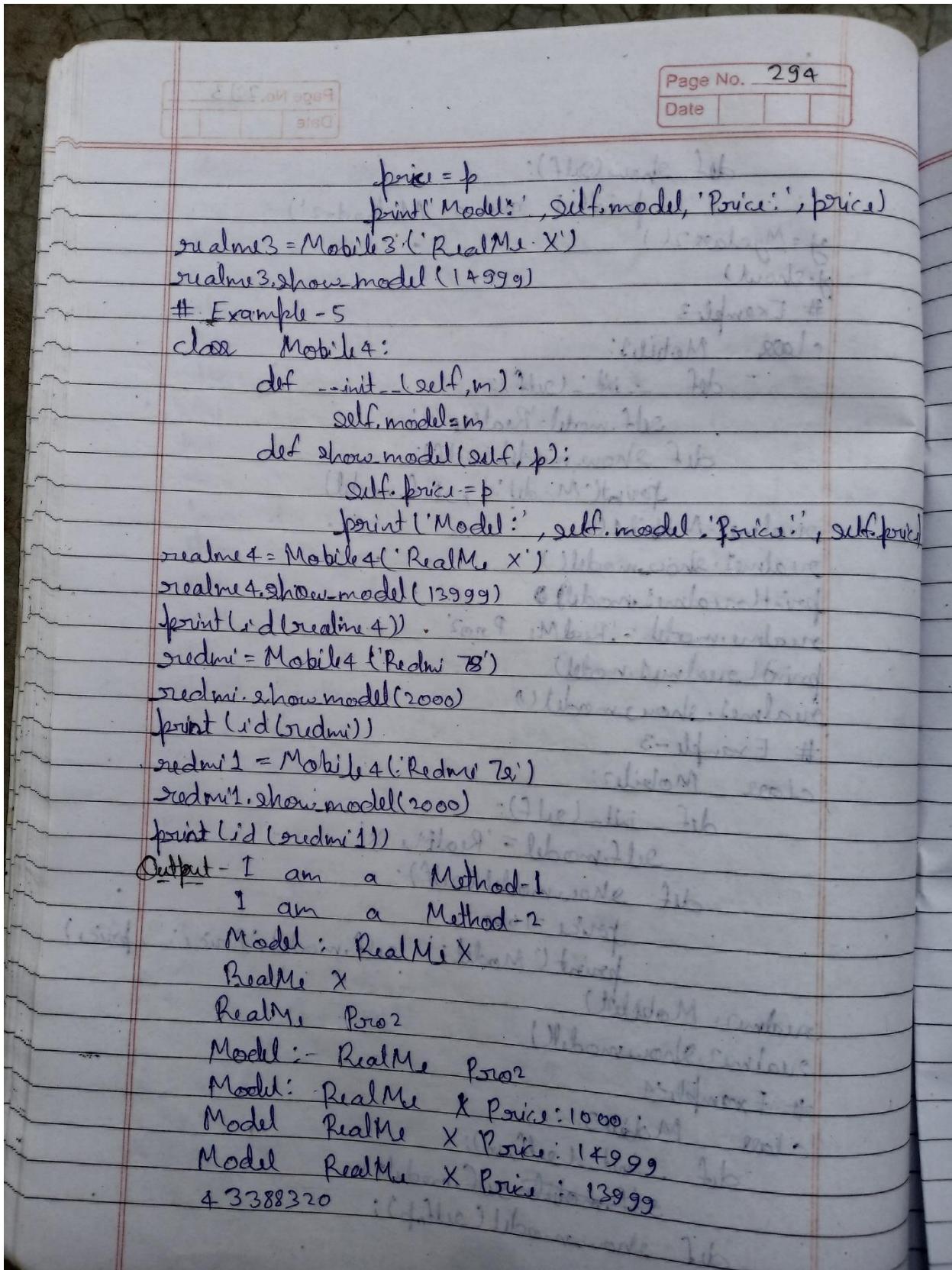
Ex- (i) def \_\_init\_\_(self):  
(ii) def showmodel(self):

→ # Example-1

```

class MyClass1(object):
    def show(self):
        print('I am a Method-1')
x=MyClass1()
x.show()
# Example-2
class MyClass2:
```





Page No. 295  
Date

Model: Redmi 7a Price: 2000  
43388208

Model: Redmi 7a Price: 2000  
43388376

Constructor -

- Python supports a special type of method constructor for initializing the instance variable of a class.
- A class constructor, if defined is called whenever whenever a program creates an object of that class.
- A constructor is called only once at the time of creating an instance. If two instances are created for a class, the constructor will be called once for each instance.

Constructor without Parameters -

Ex- class Mobile:

```
def __init__(self):
    self.model = 'RealMe X'
```

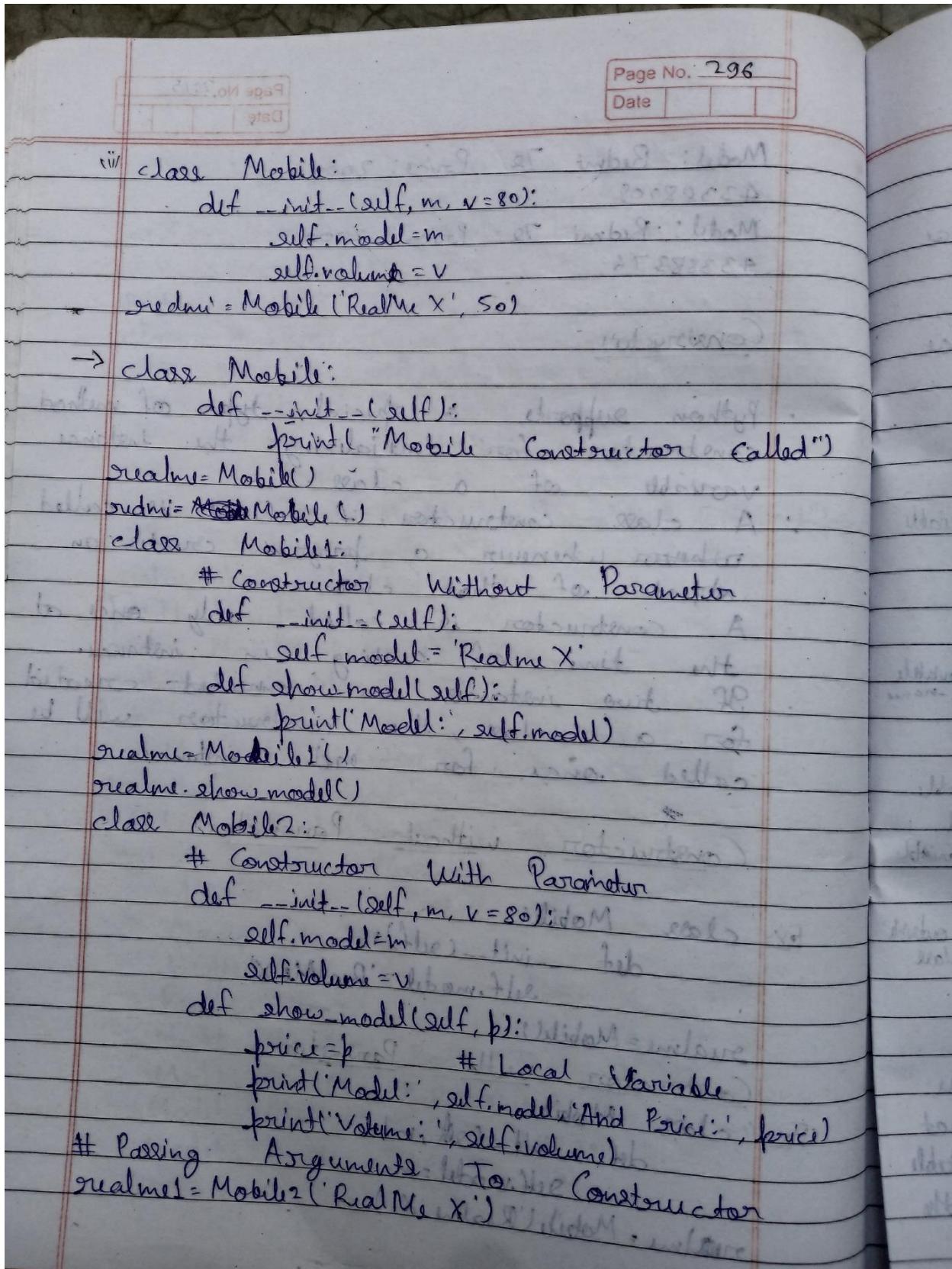
realme = Mobile()

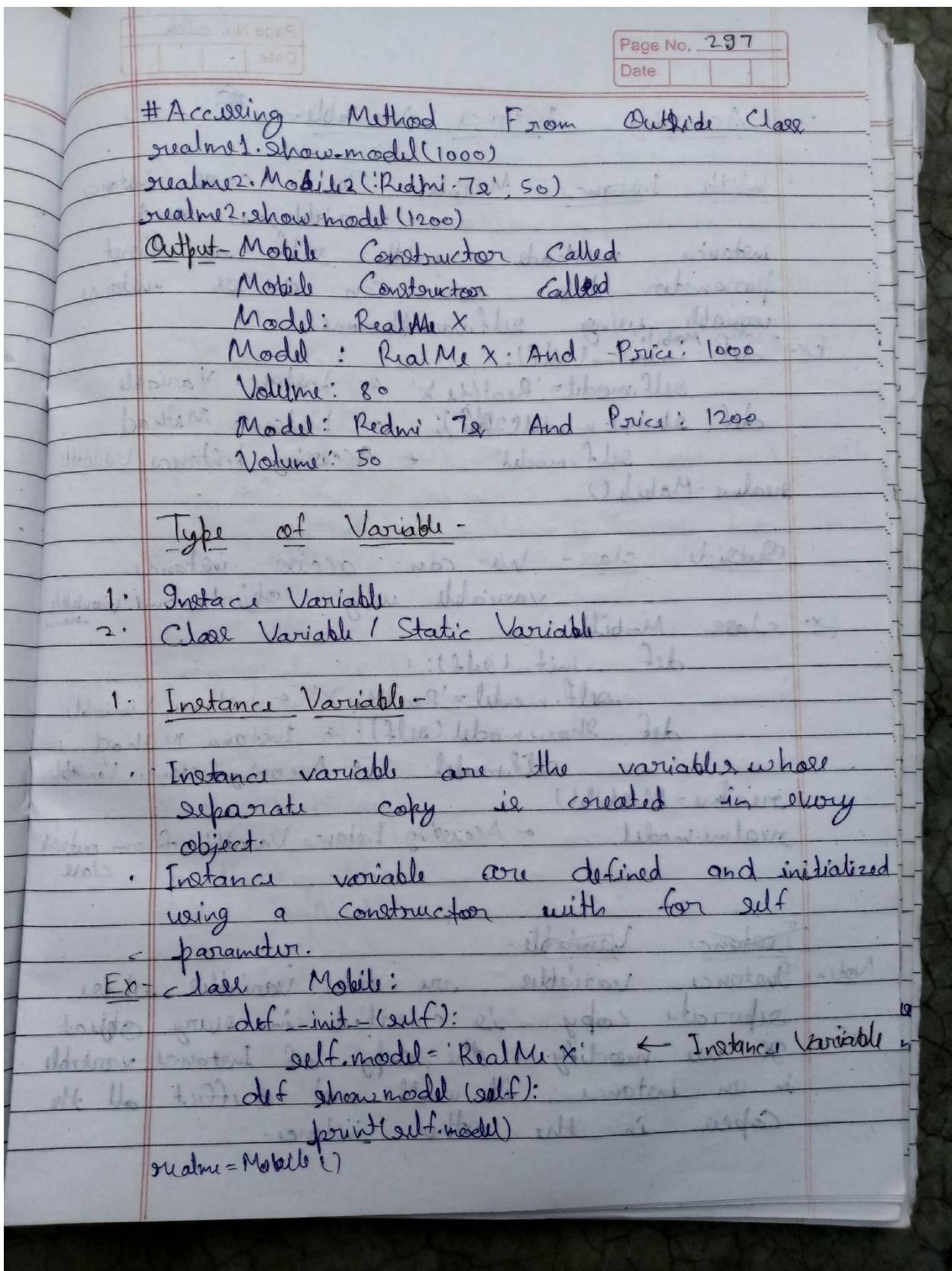
Constructor with Parameters -

Ex- " class Mobile:

```
def __init__(self, m):
    self.model = m
```

realme = Mobile('Realme X')





Page No. 298  
Date

### Accessing Instance Variable - Q.

- With Instance Method - To access instance variable, we need instance methods with self as first parameter then we can access instance variable using self.variableName.

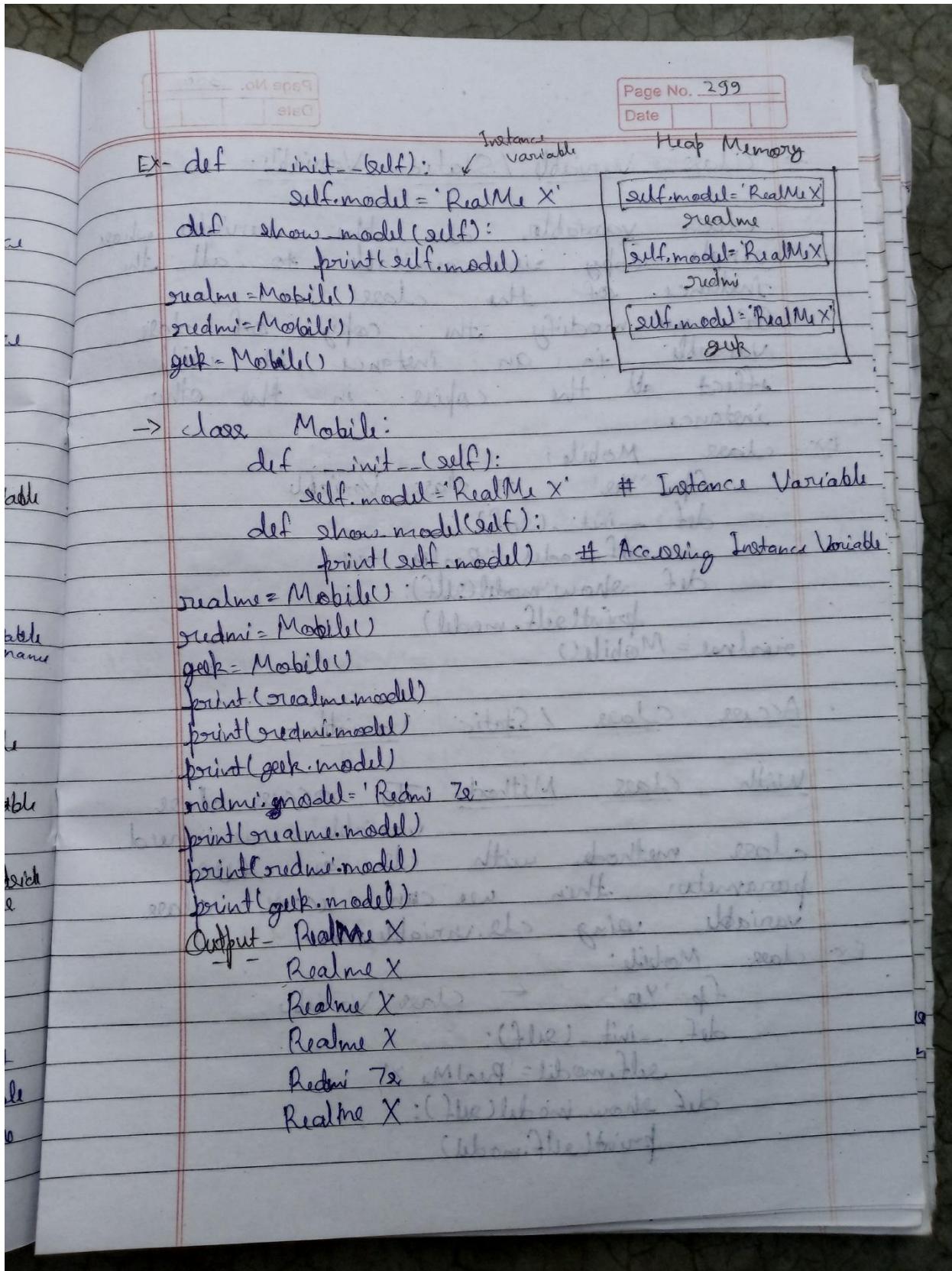
Ex- `class Mobile:`  
 `def __init__(self):`  
 `self.model = 'RealMe X' ← Instance Variable`  
 `def showModel(self): ← Instance Method`  
 `self.model ← Accessing Instance Variable`  
`realme = Mobile()`

- Outside class - We can access instance variable using object.name.variableName

Ex- `class Mobile:`  
 `def __init__(self):`  
 `self.model = 'RealMe X' ← Instance Variable`  
 `def showModel(self): ← Instance Method`  
 `self.model ← Accessing Instance Variable`  
`realme = Mobile()`  
`realme.model ← Accessing Instance Variable from outside class`

### Instance Variable -

Note - Instance variables are the variables whose separate copy is created in every object. If we modify the copy of instance variable in one instance, it will not affect all the copies in the other instances.



Page No. 305  
Date \_\_\_\_\_

2. Class Variable / Static Variable -

- class variables are the variables which single copy ie available to all the instances of the class.
- If we modify the copy of class variable in an instance, it will affect all the copies in the other instances.

Ex- class Mobile:

```

fp = 'Yes'           ← class variable
def __init__(self):
    self.model = 'RealMe X'
def show_model(self):
    print(self.model)
realme = Mobile()

```

Access Class / Static Variable -

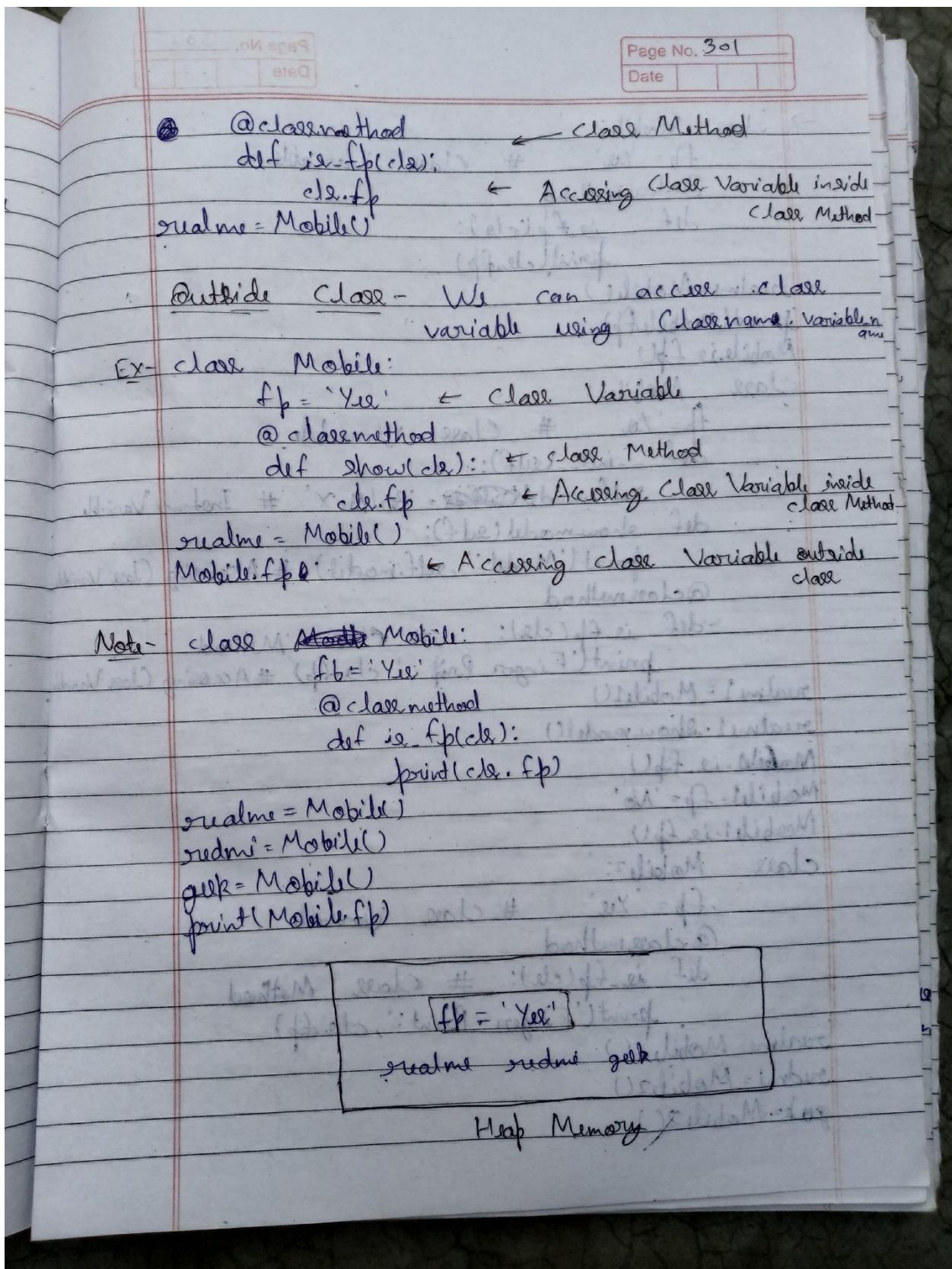
With Class Method: To access class variable, we need class methods with `cls` as first parameter then we can access class variable using `cls.variable_name`.

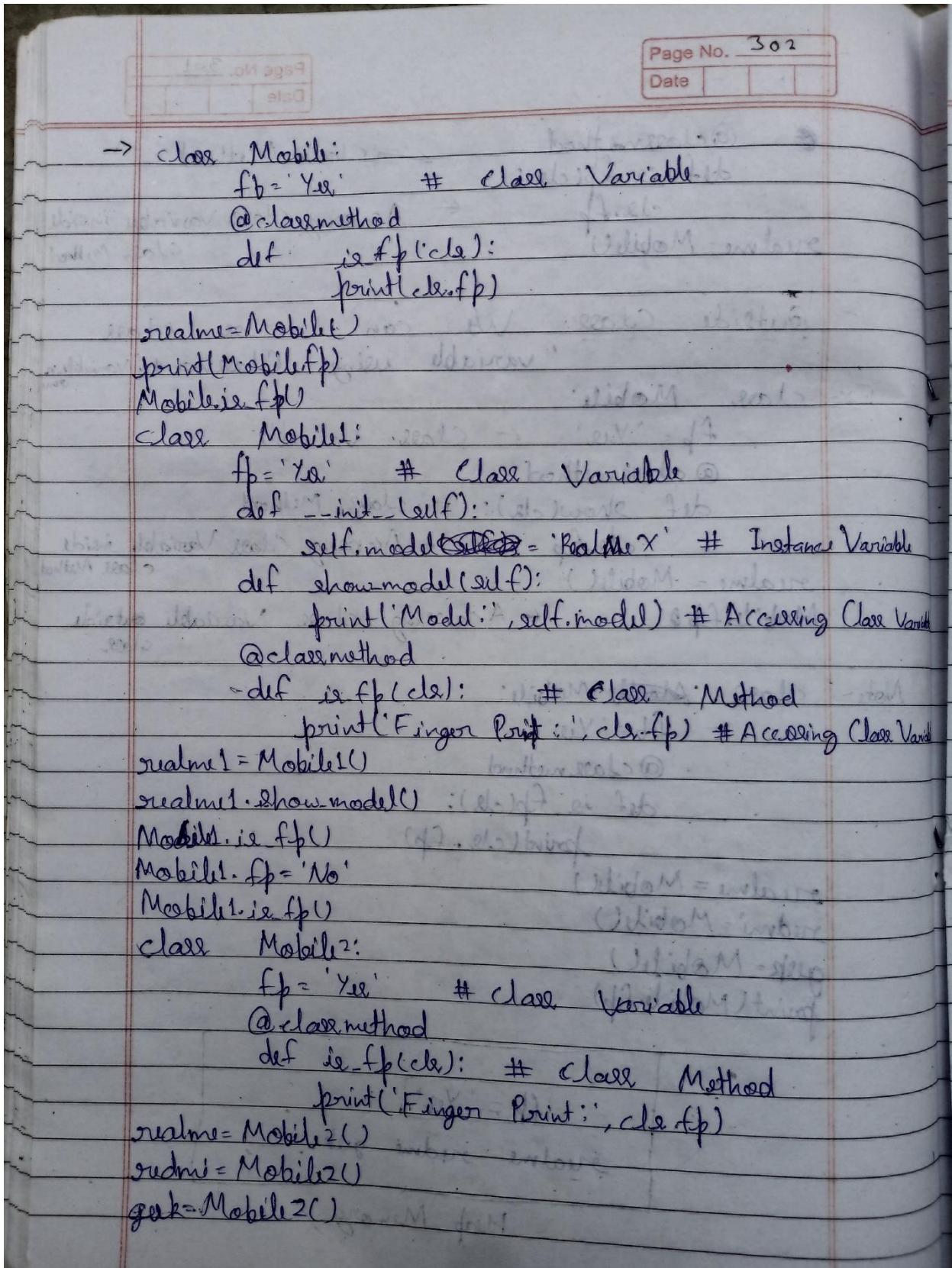
Ex- class Mobile:

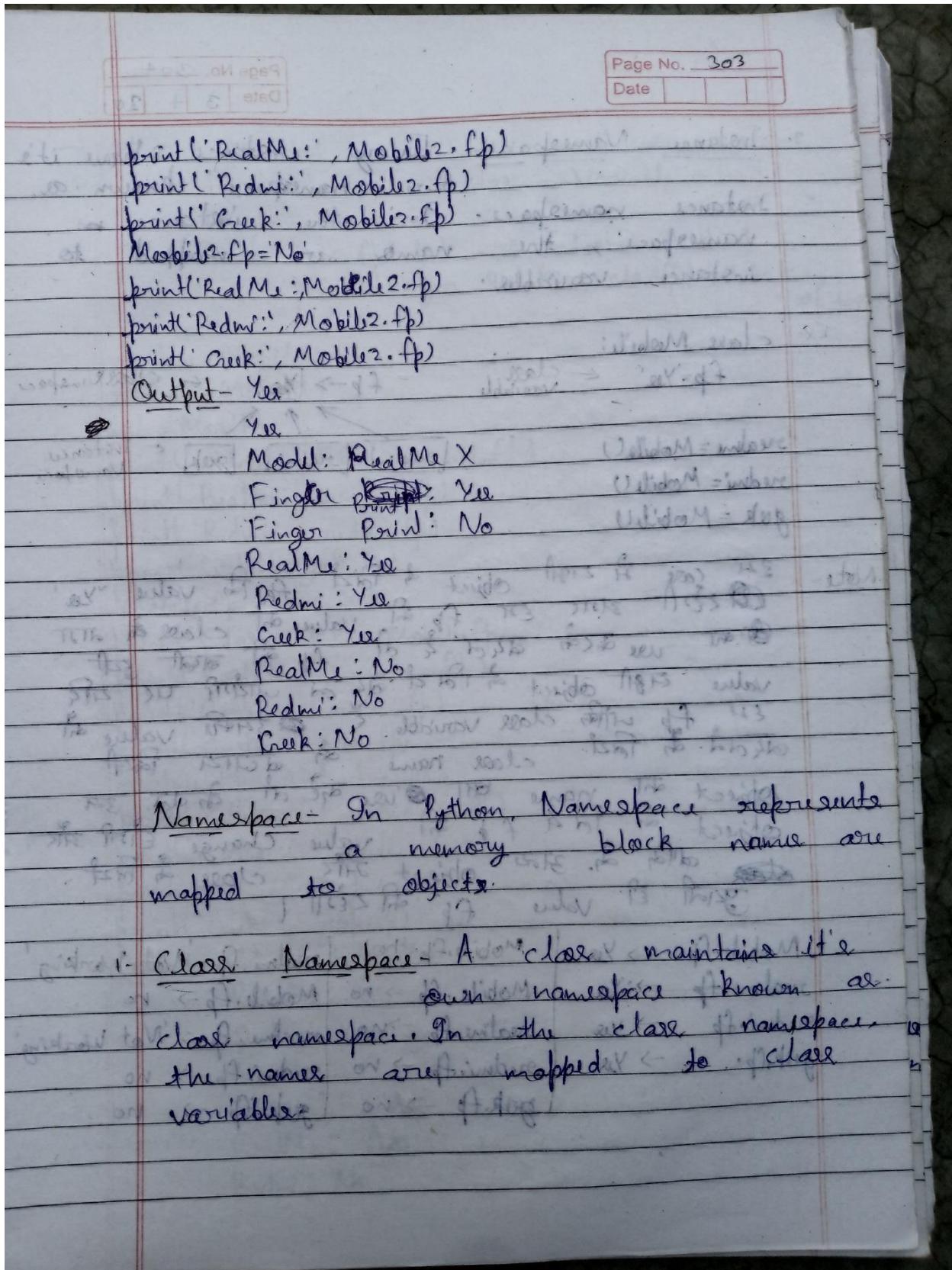
```

fp = 'Yes'           ← class variable
def __init__(self):
    self.model = 'RealMe X'
def show_model(self):
    print(self.model)

```







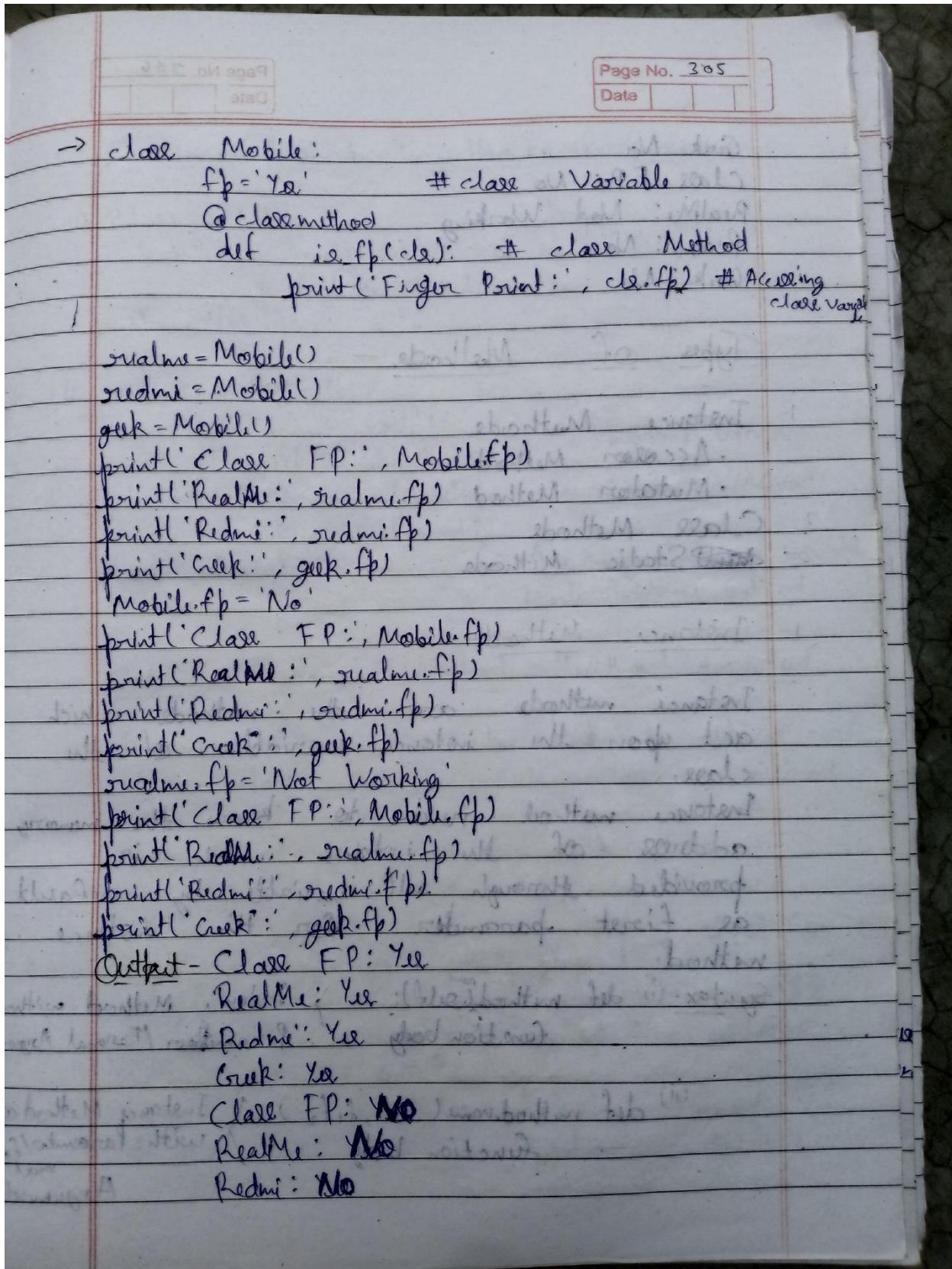
Page No. 304  
Date 3 4 20

2. Instance Namespace - Every instance have its own namespace known as instance namespace. In the instance namespace, the names are mapped to instance variables.

Ex: class Mobile:  
 $fp = 'ya'$  ← class variable       $fp \rightarrow [\underline{ya}]$  ← class namespace  
 realme = Mobile()       $\underline{\text{realme}}$        $\underline{\text{redmi}}$        $\underline{\text{geek}}$  ← Instance Namespace  
 redmi = Mobile()  
 geek = Mobile()

Note:- ~~जब जैसे object के लिए fp का value "ya"~~  
~~जैसे ही fp की value को class को नहीं~~  
~~किया तो करके बदल देते हैं तो fp की वाले हुए~~  
~~value वही object के लिए बदल जायगी पर यहाँ~~  
~~इस fp with class variable के उसकी value की~~  
~~बदलने के लिए class name को बदला दिये~~  
~~object का name भी बदल देते हैं के बारे उस~~  
~~object के लिए fp की value change होती और~~  
~~वही ही fp की value change होती और object के बारे~~  
~~class के लिए~~

Mobile.fp → ya	Mobile.fp = 'no'	realme.fp = 'Not Working'
realme.fp → ya	Mobile.fp → no	Mobile.fp → no
redmi.fp → ya	realme.fp → no	realme.fp → 'Not Working'
geek.fp → ya	redmi.fp → no	redmi.fp → no
	geek.fp → no	geek.fp → no



Page No. 306  
Date

Geek: No  
Class FP: No  
RealMe: Not Working  
RealMi: No  
Geek: No

## Type of Methods

- 1- Instance Method
  - Accessor Method
  - Mutator Method
- 2- Class Method
- 3- ~~and~~ Static Methods

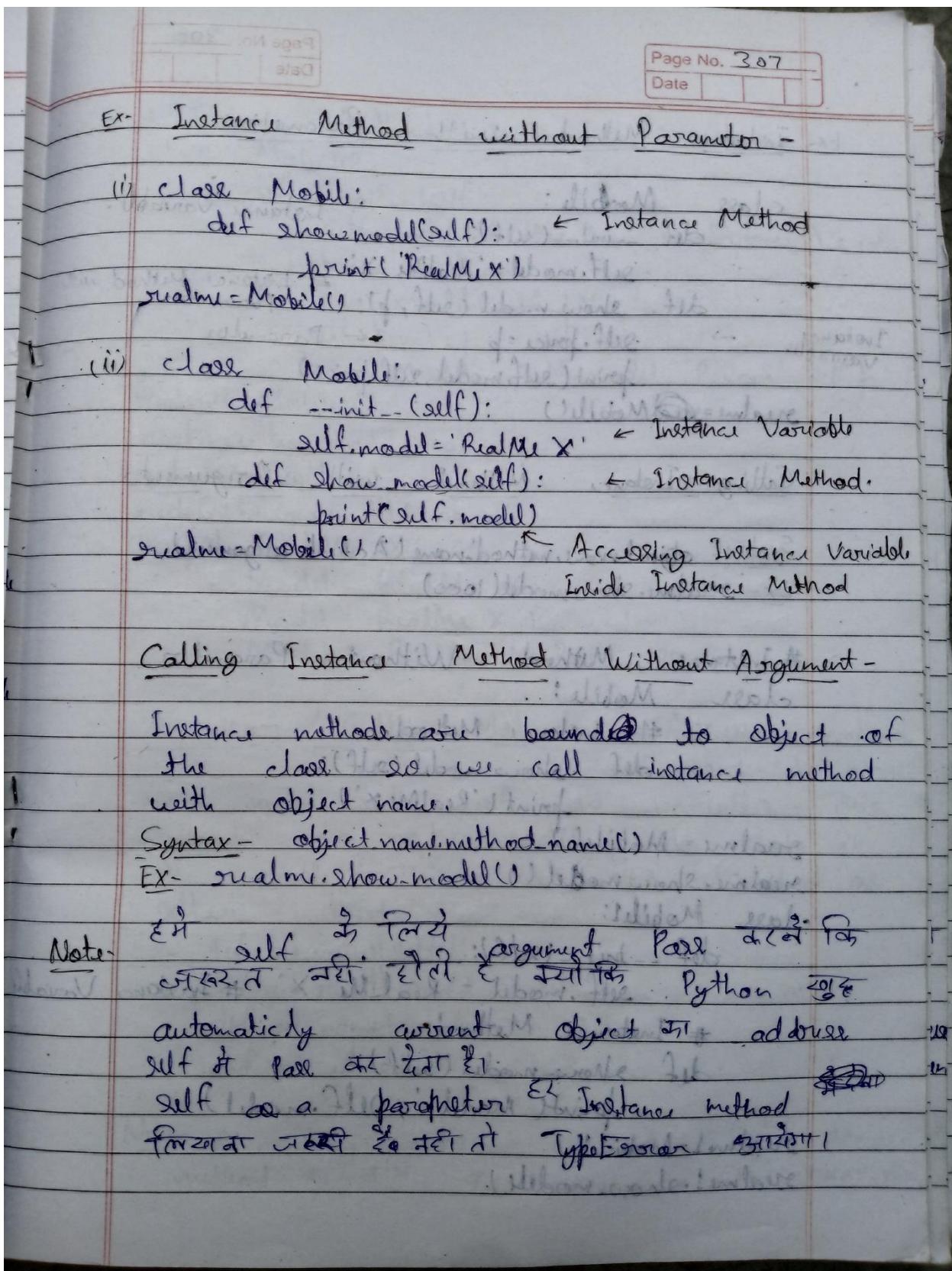
### Instance Method

Instance methods are the methods which act upon the instance variables of the class.

Instance method need to know the memory address of the instance which is provided through self variable by default as first parameter for the instance method.

Syntax- (i) `def method(nameself):` } Instance Method without function body      } Parameter / Formal Argument

(ii) `def method.name(self, f1, f2, ...):` } Instance Method with Parameters / Actual Arguments  
                function body : {



Page No. 308  
Date

Ex- Instance Method with Parameter-

class Mobile:

```

    def __init__(self):
        self.model = 'RealMe X' # Instance Variable.

    def show_model(self, p):
        self.price = p # Parameter
        print(self.model, self.price)

```

realme = Mobile() # Instance Variable.

Calling Instance Method with Argument -

Syntax- objectname.method-name (Actual argument)

Ex- realme.show\_model(1000)

→ # Instance Method Without Parameter

class Mobile:

```

    def show_model(self):
        print('RealMe X')

```

realme = Mobile() # Instance Method without Parameter

realme.show\_model() # Instance Method without Parameter

class Mobile:

```

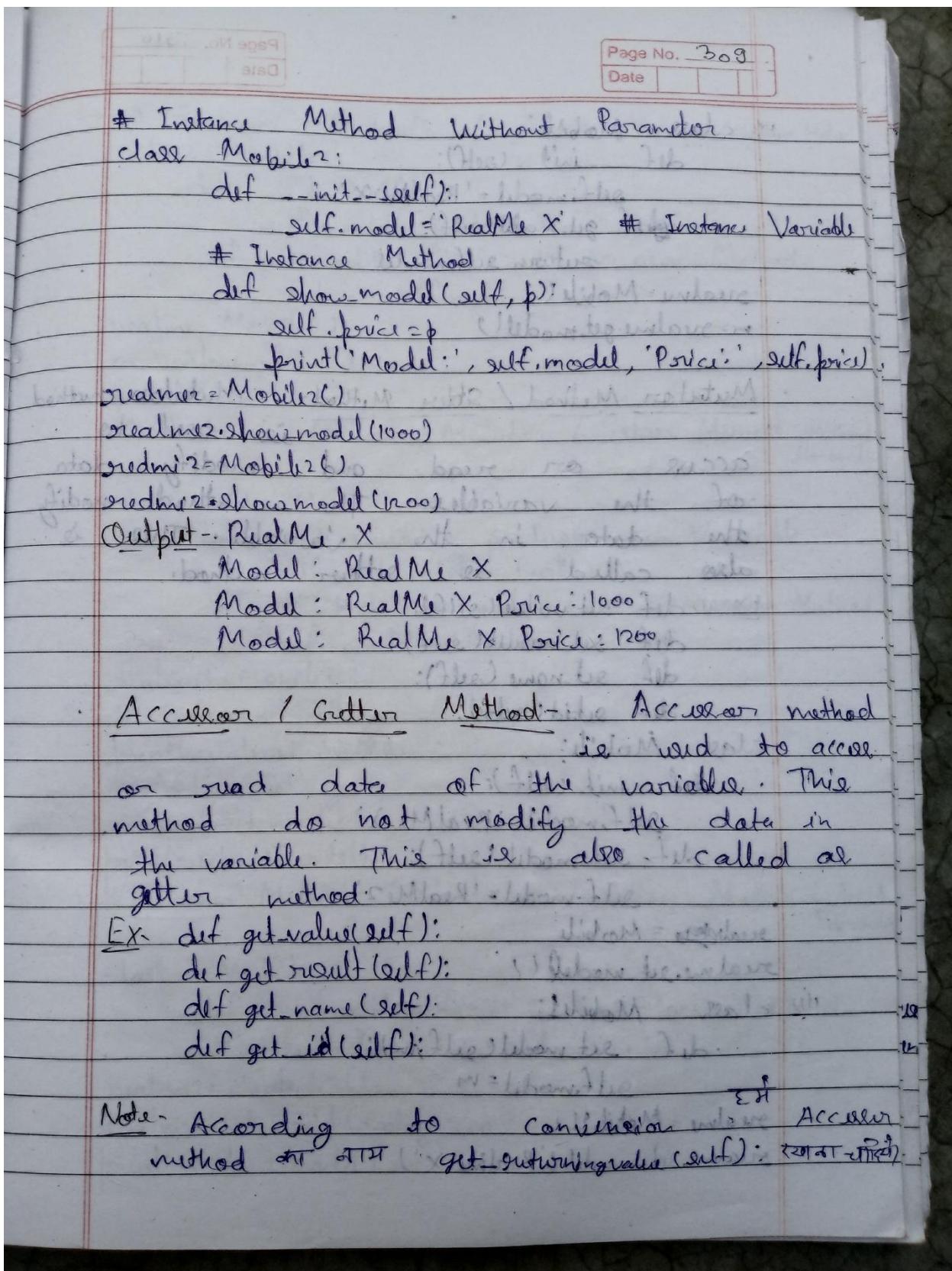
    def __init__(self):
        self.model = 'RealMe X' # Instance Variable.

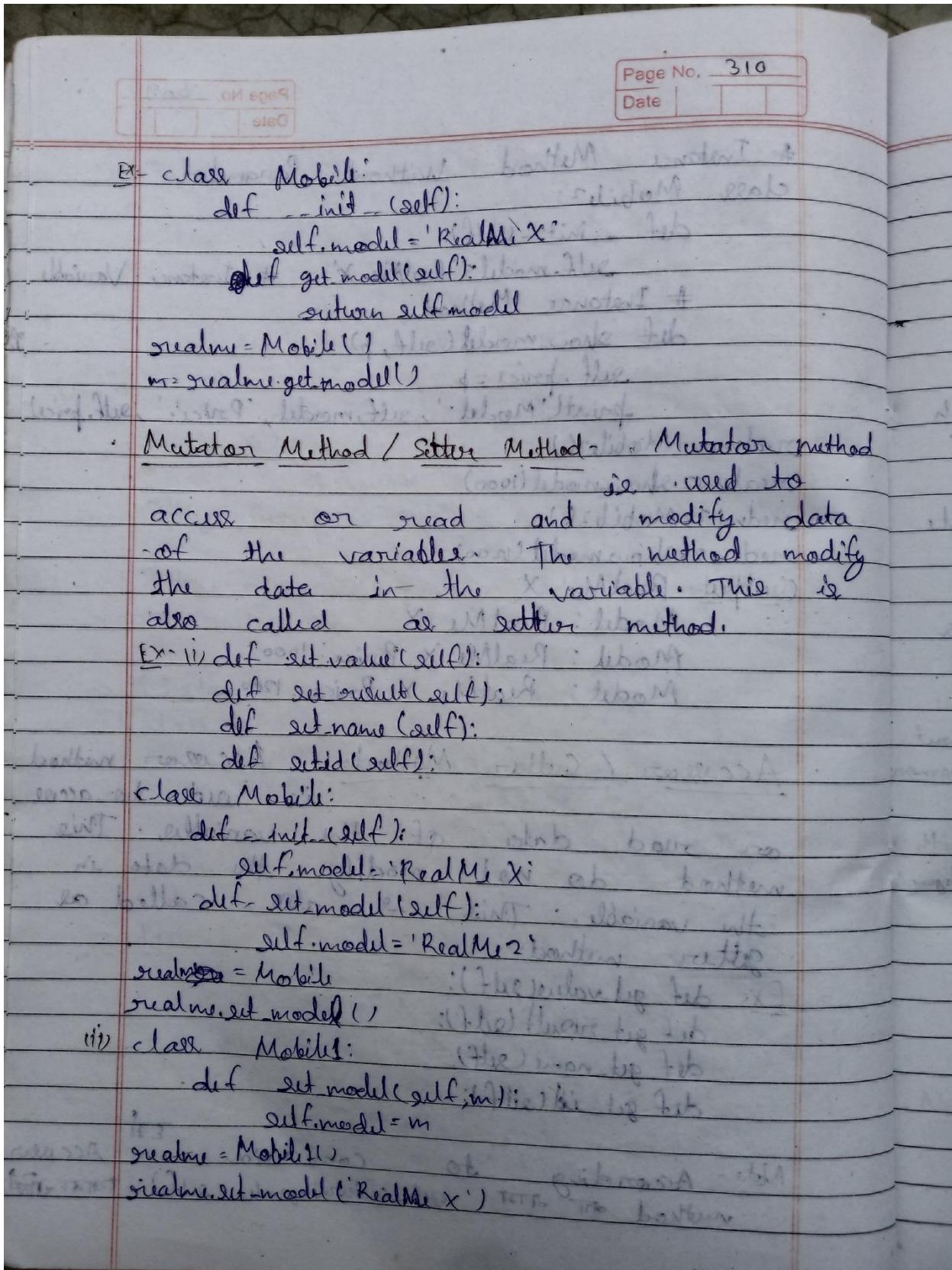
    def show_model(self):
        print('Model:', self.model)

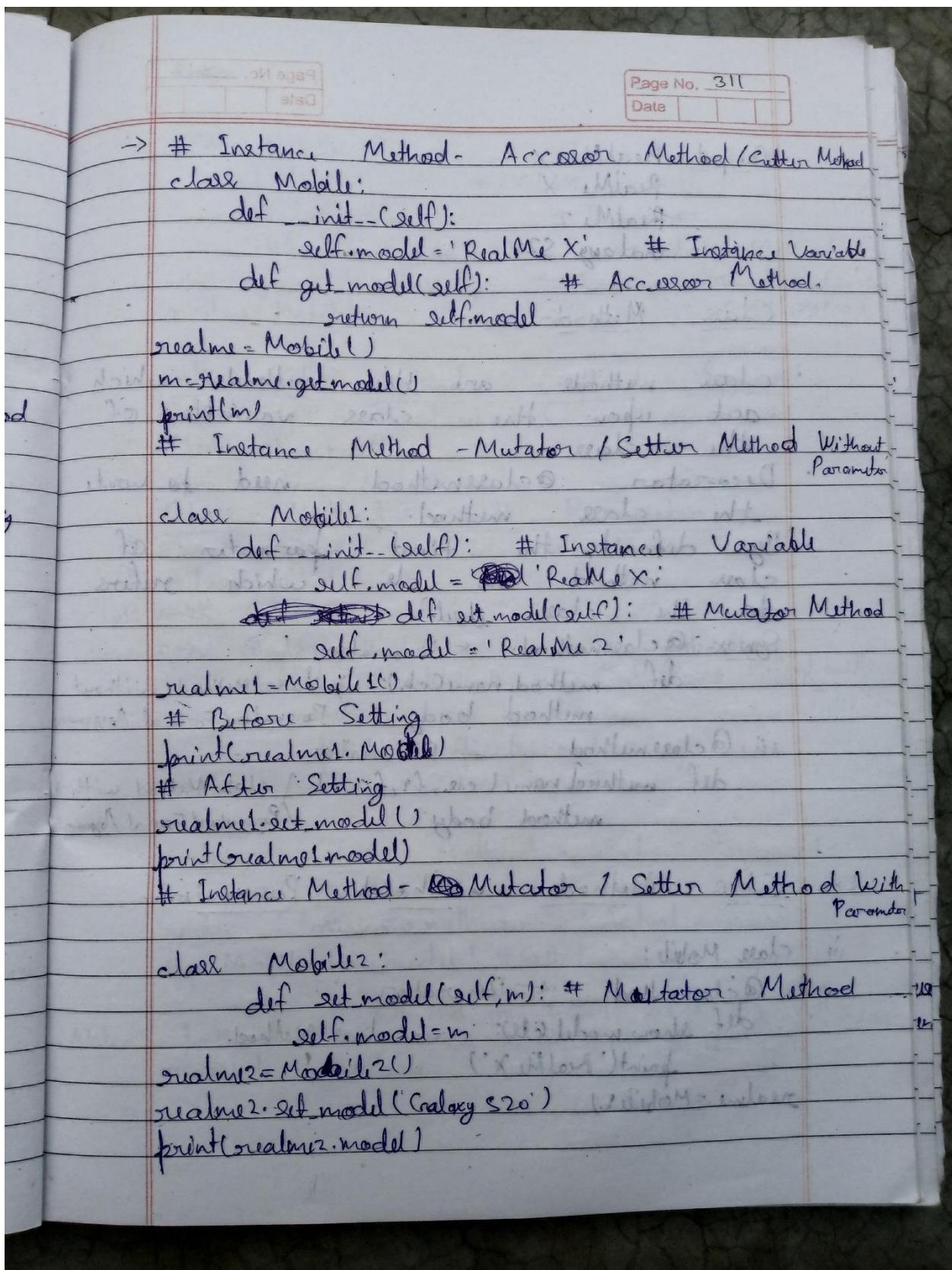
```

realme1 = Mobile() # Instance Method without Parameter

realme1.show\_model() # Instance Method without Parameter







Page No. 312  
Date

Output - RealMe X

RealMe X

RealMe 2

Galaxy S20

2- Class Methods

- class methods are the methods which act upon the class variable of the class.
- Decorator `@classmethod` need to rewrite the class method.
- By default, the first parameter of class method is `cls`, which refers to the class itself.

Syntax :- `@classmethod` ← Decorator

```

def method_name(cls):    } class Method without
                        } Parameter/ Formal Argument
method body
  
```

(i) `@classmethod` ← Decoration

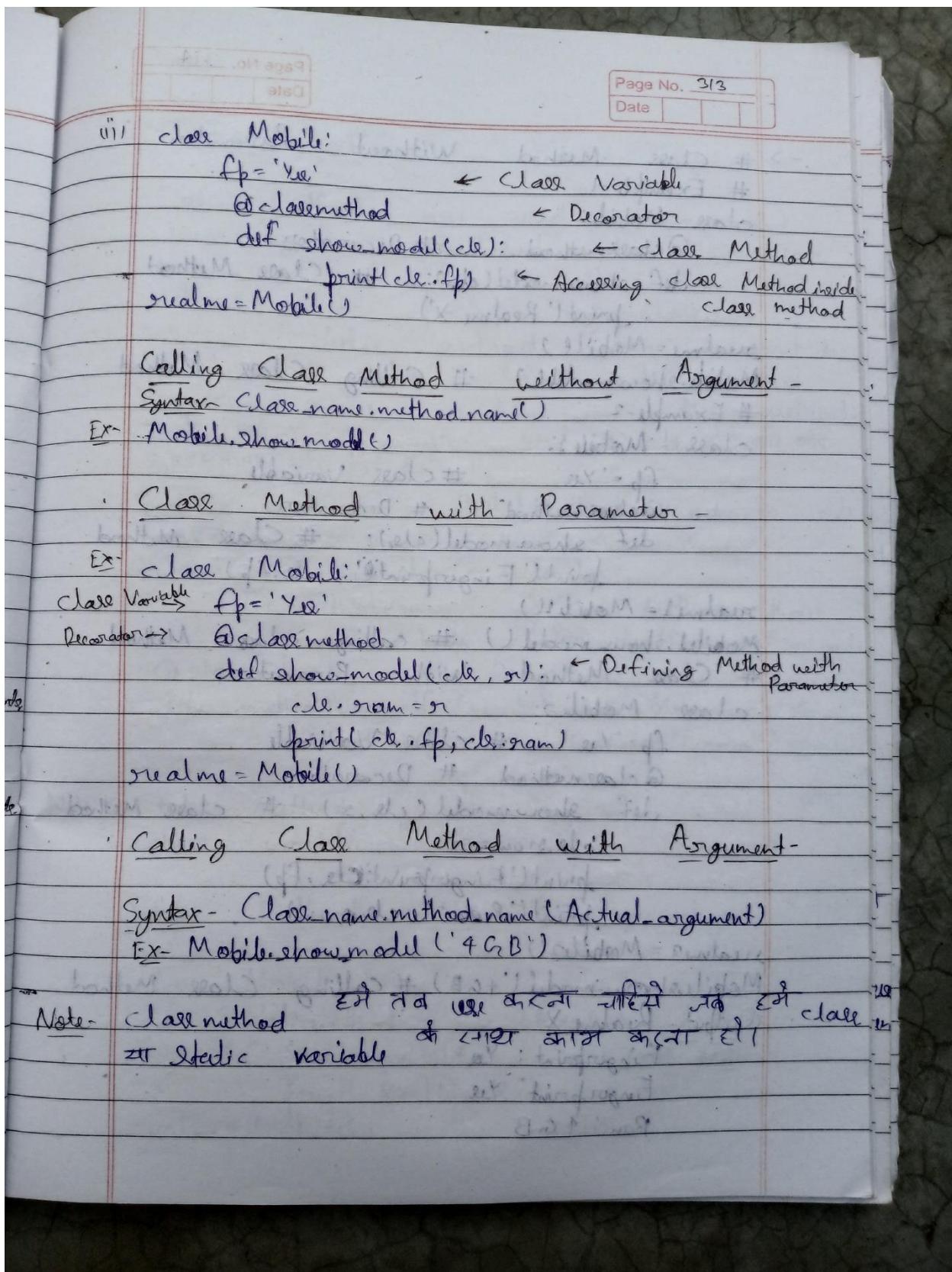
```

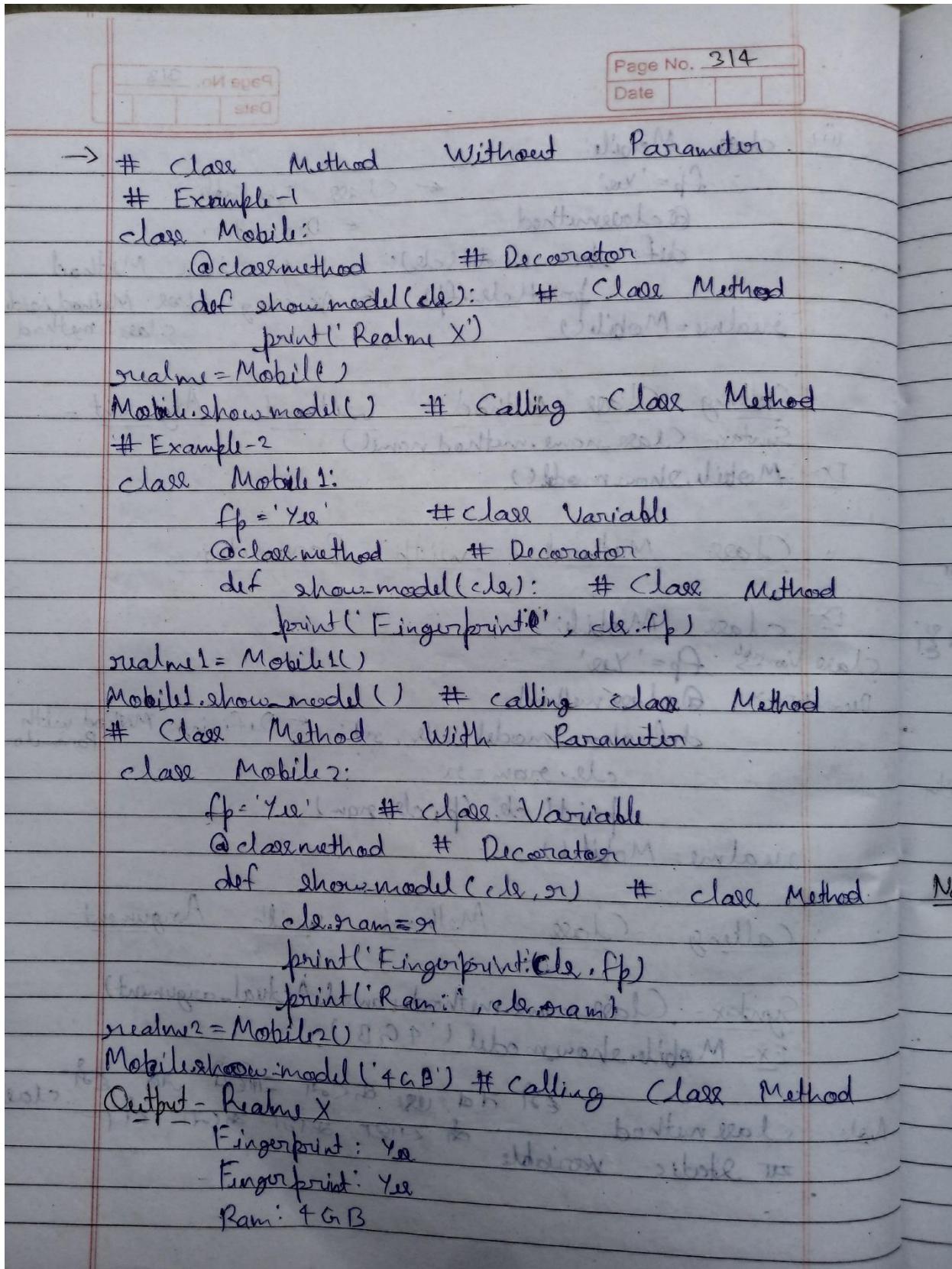
def method_name(cls, f1, f2):    } class Method with
                                } Parameter/ Formal Argument
method body
  
```

Ex- Class Method Without Parameter -

```

in class Mobile:
    @classmethod    } ← Decorator
    def showModel(cls):    } ← class Method.
        print('RealMe X')
    realme = Mobile()  } (as per question) show for realme
  
```





Page No. 215  
Date

### 3. Static Methods

- Static Methods are used when some processing is related to the class but does not need the class or its instance to perform any work.
- We use static method when we want to pass some values from outside and perform some action in the method.
- परंतु पैसी परामिटर self नहीं होते हैं।  
प्रैमिटर नहीं होते हैं।
- ~~Symbol~~ Decorator @staticmethod need to write above the static method.

Syntax - `@staticmethod ← Decorator`

```

def method_name(): } Static Method without
                  method body } Parameter/Formal Arguments
    @staticmethod ← Decorator
    def method_name(f1, f2, ...): } Static Method with
        method body } Parameter/Formal Arguments
  
```

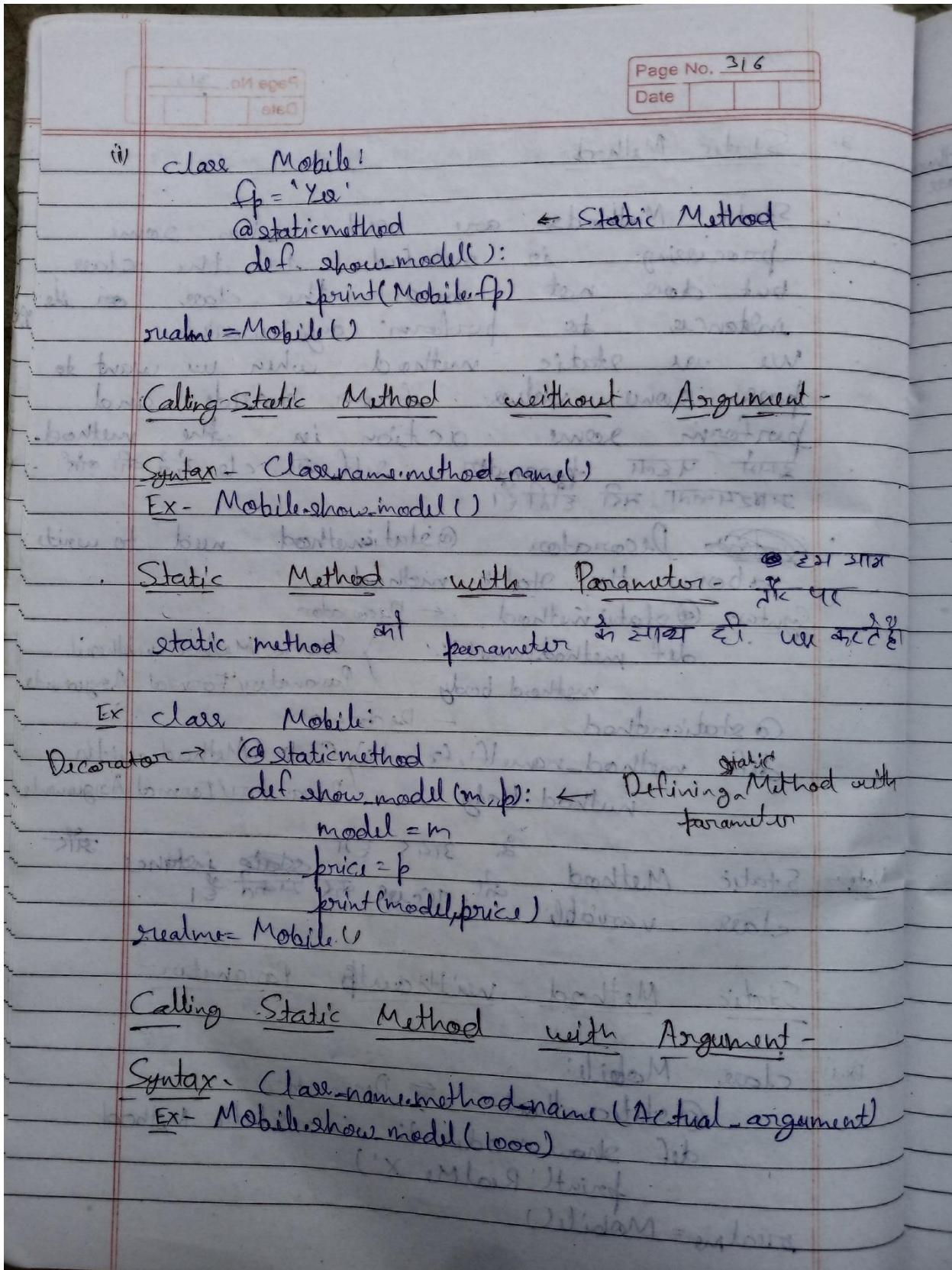
Note - Static Method <sup>जो बीज एवं</sup> <sup>static instance के</sup>  
 class variable <sup>का</sup> <sup>प्रैसेस कर सकते हैं।</sup>

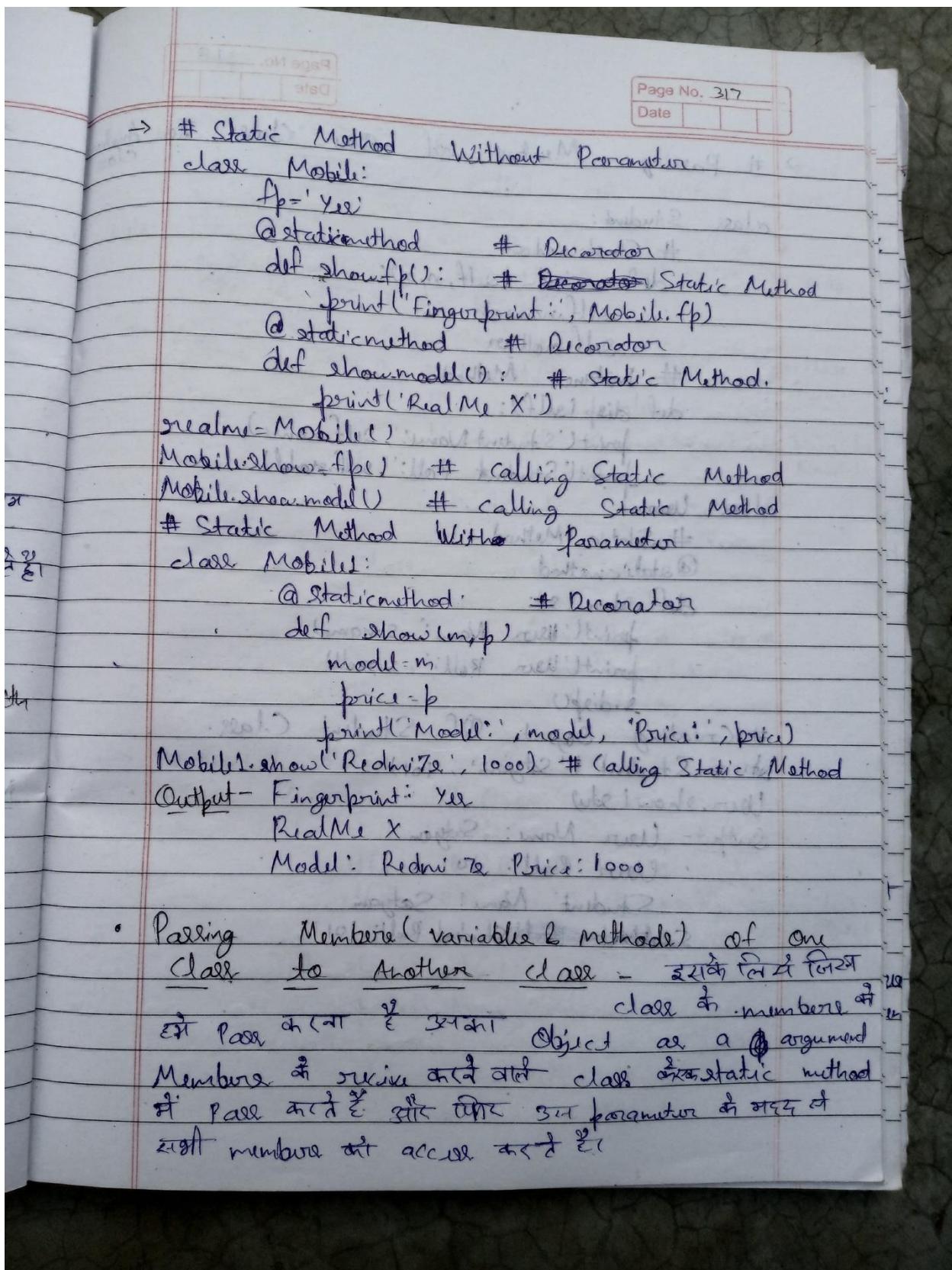
### Static Method without Parameter

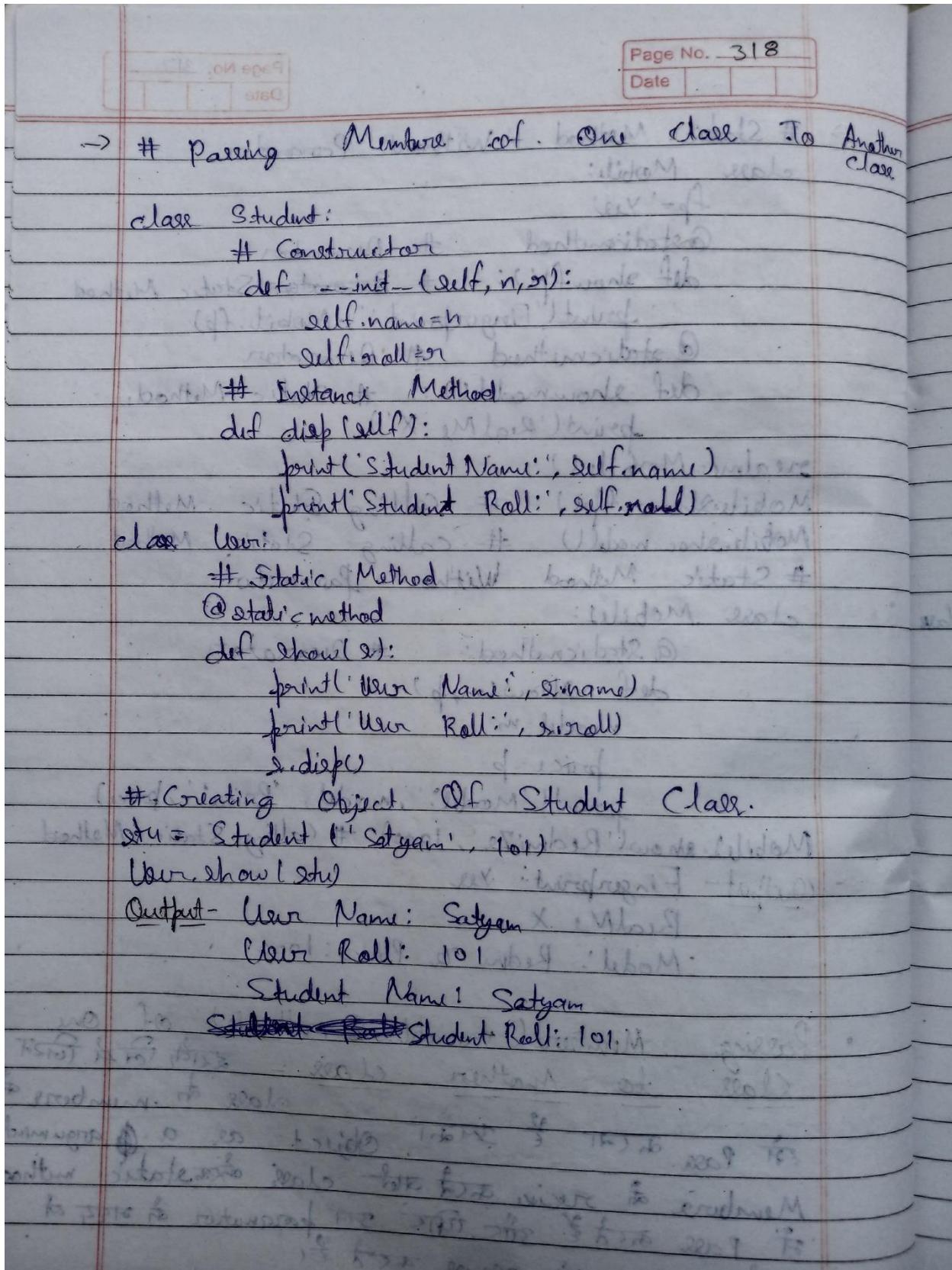
Ex-iv) `class Mobile:`

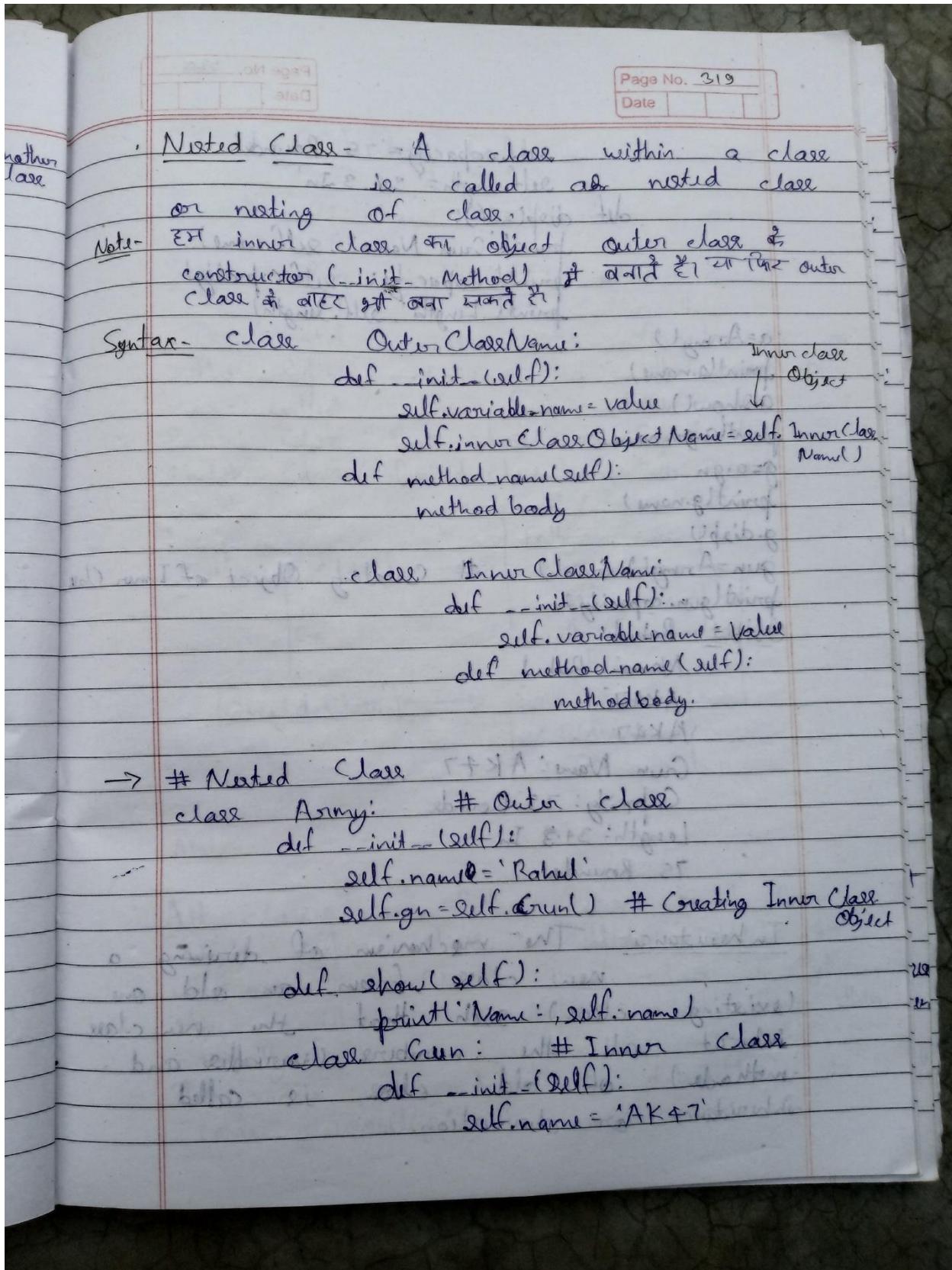
```

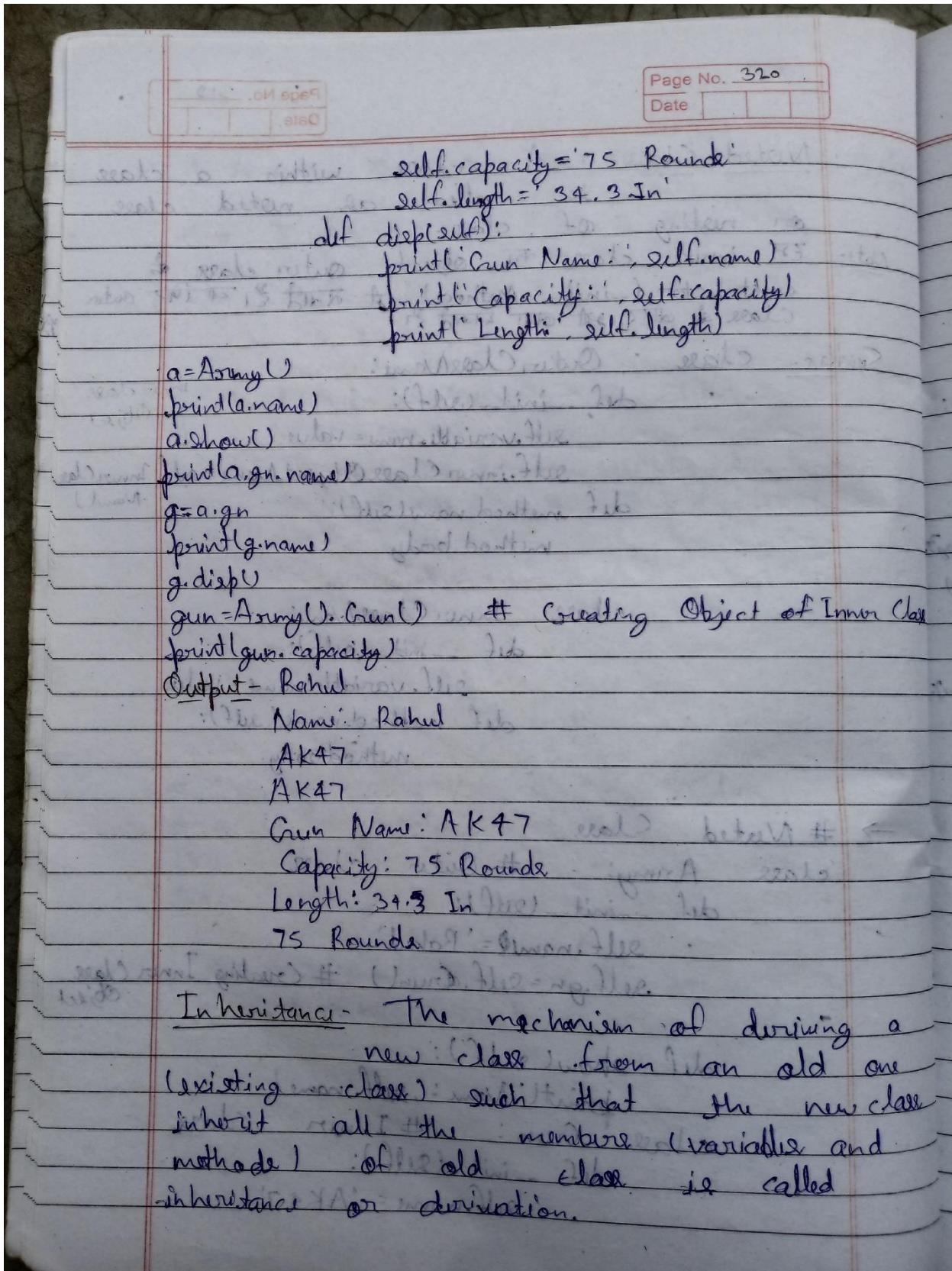
@staticmethod ← Decorator
def show_model(): } ← Static Method
    print('RealMe X')
realMe = Mobile()
  
```

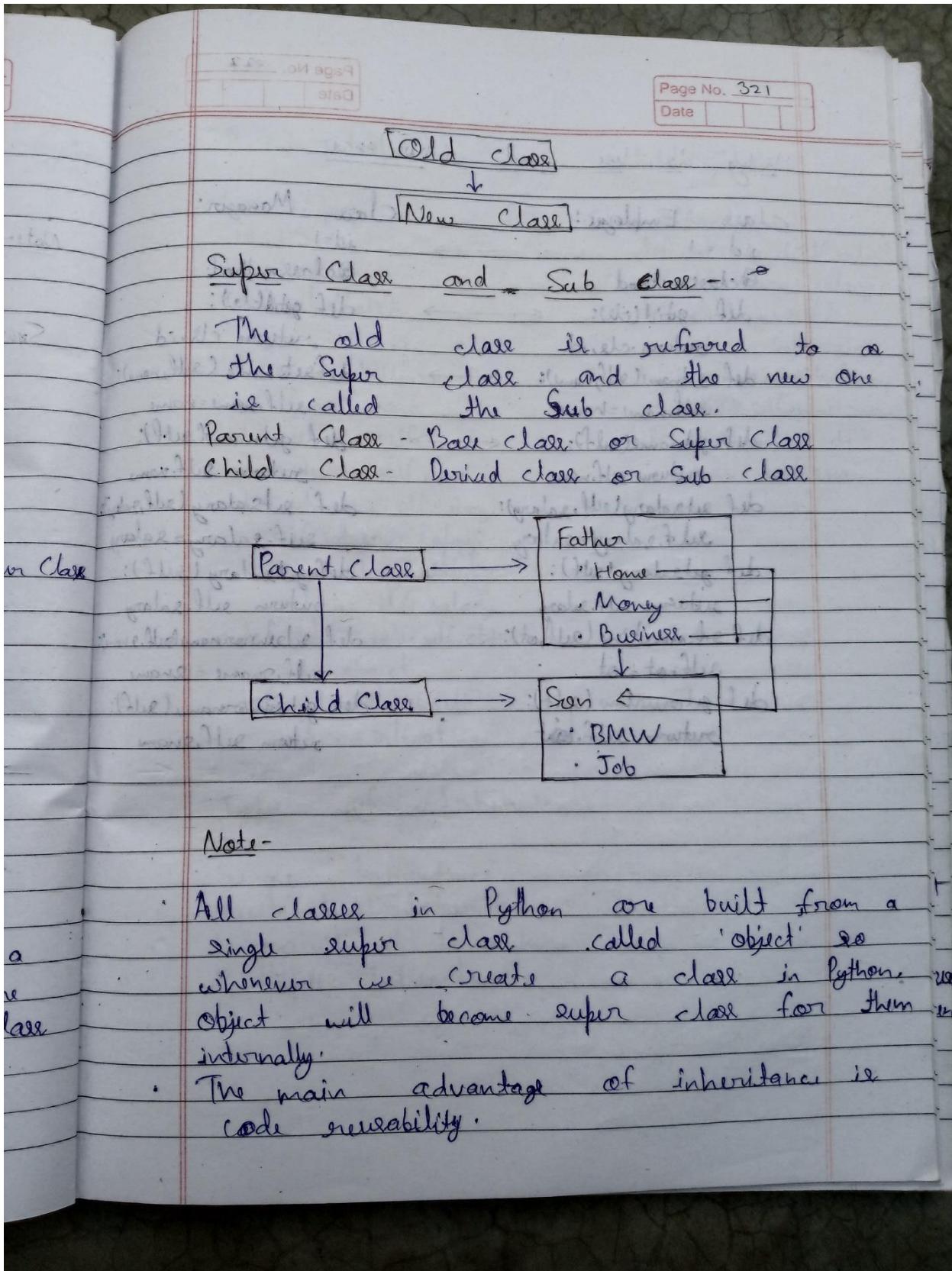


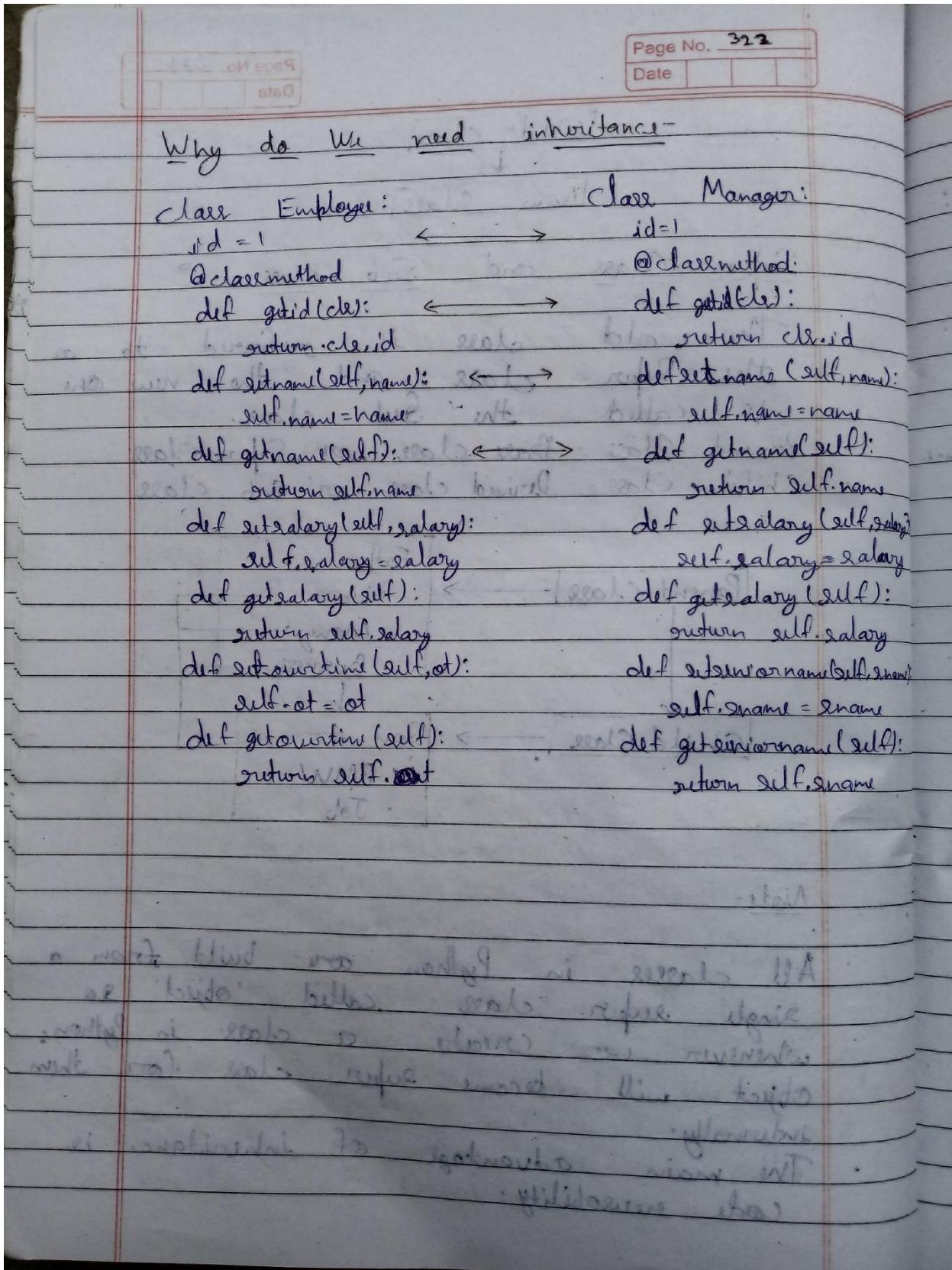


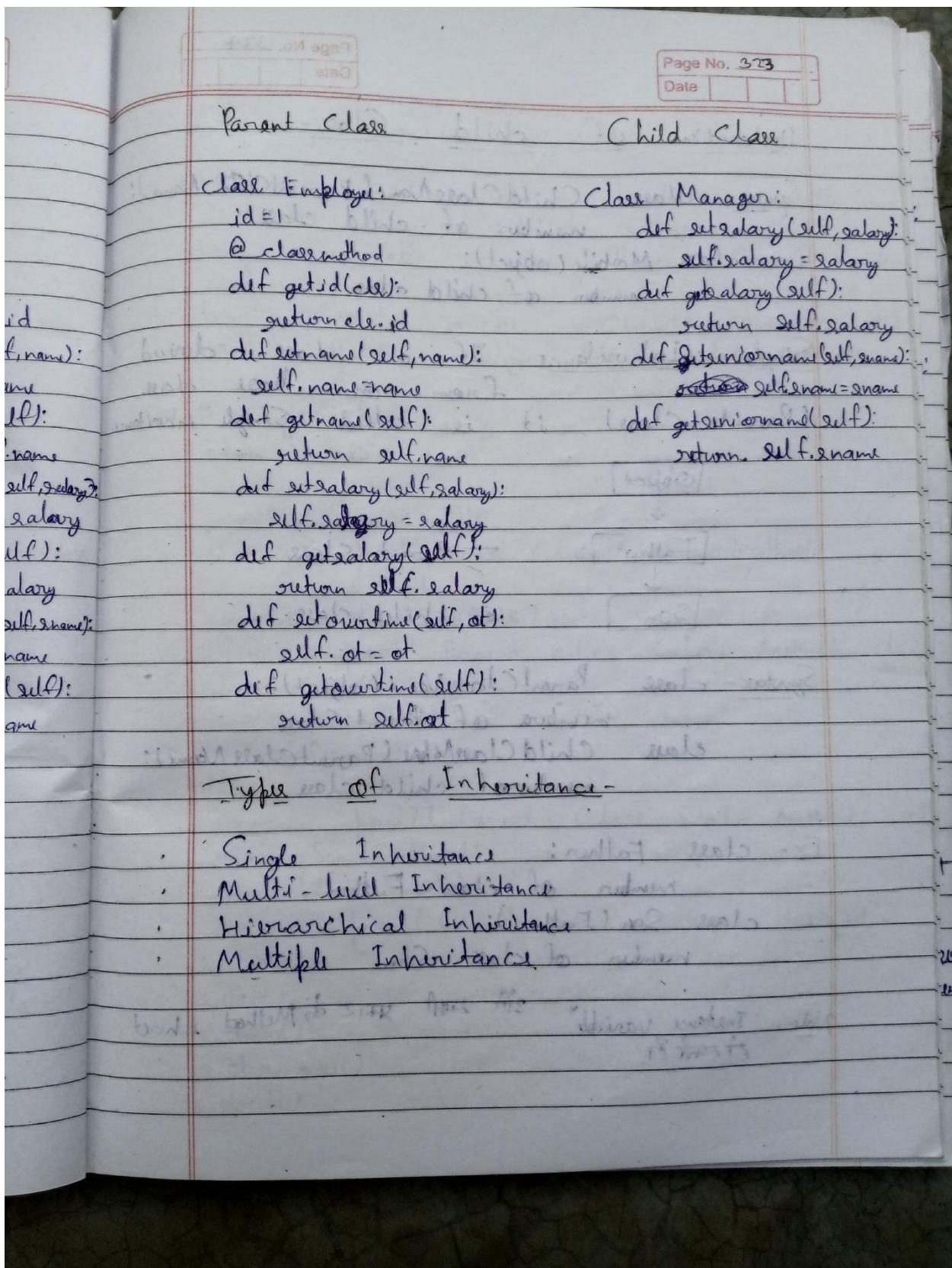












Page No. 324  
Date

Declaration of child Class -

Syntax- class ChildClassName ( ParentClassName ):  
 member of child class.

Ex- class Mobile ( object ):  
 member of child class.

Single Inheritance - If a class is derived from one base class (Parent Class), it is called Single inheritance.

```

graph TD
    Object[Object] --> Father[Father]
    Father --> Son[Son]
    Father -- "Parent class" --> Son
    Son -- "child class" --> Son
  
```

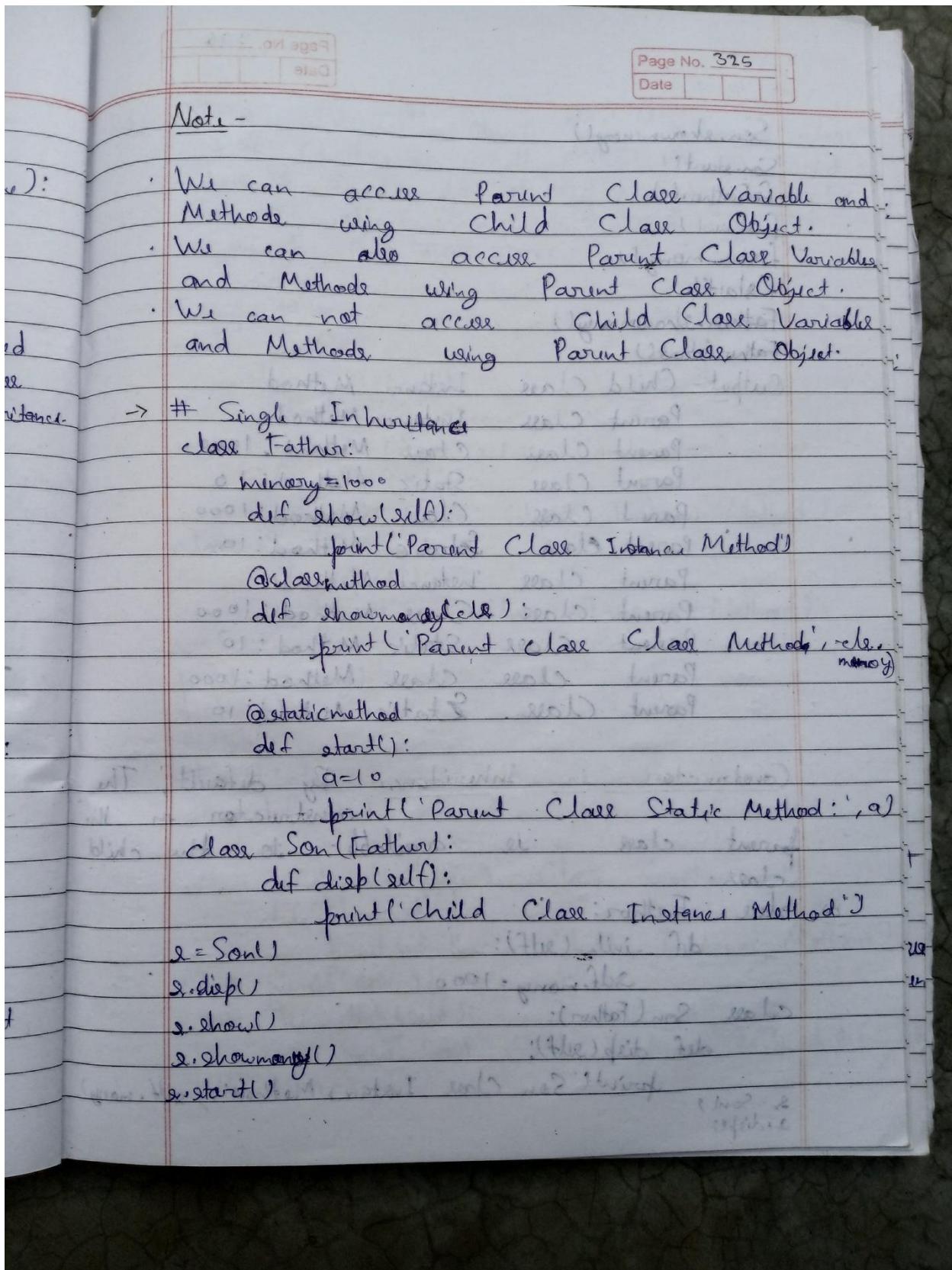
Syntax- class ParentClassName ( object ):  
 member of Parent class

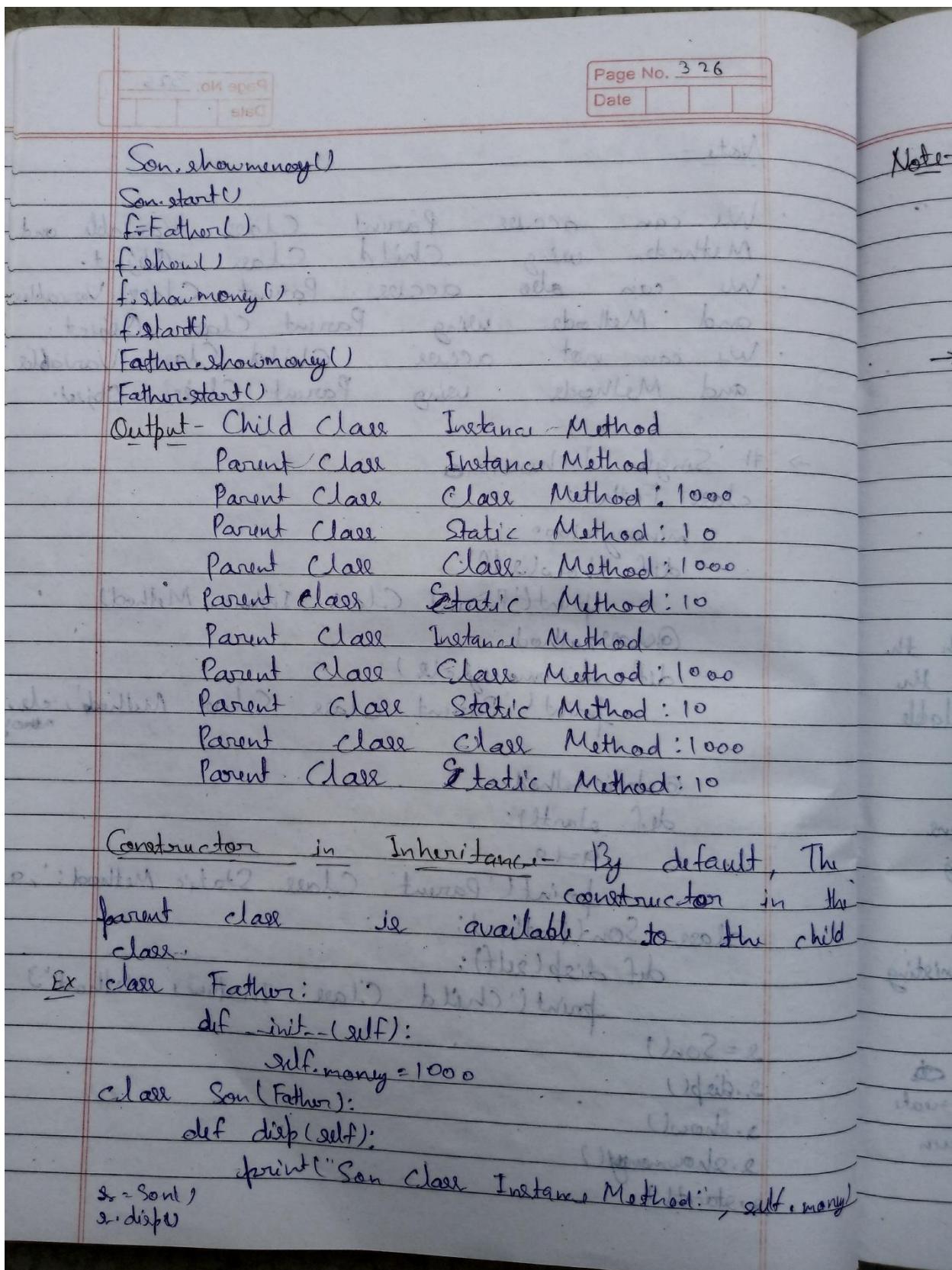
class ChildClassName ( ParentClassName ):  
 member of child class

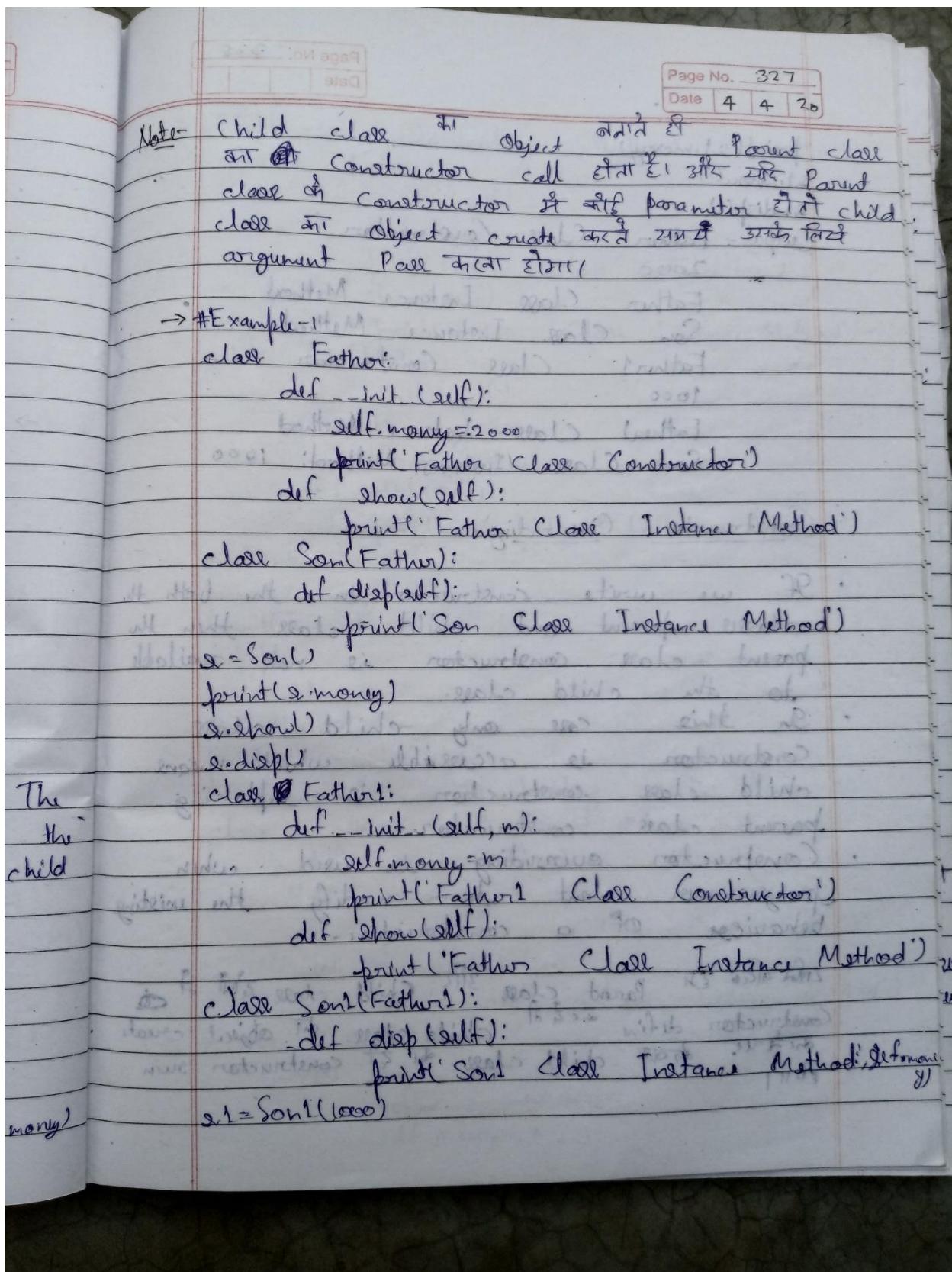
Ex- class Father:  
 member of class Father

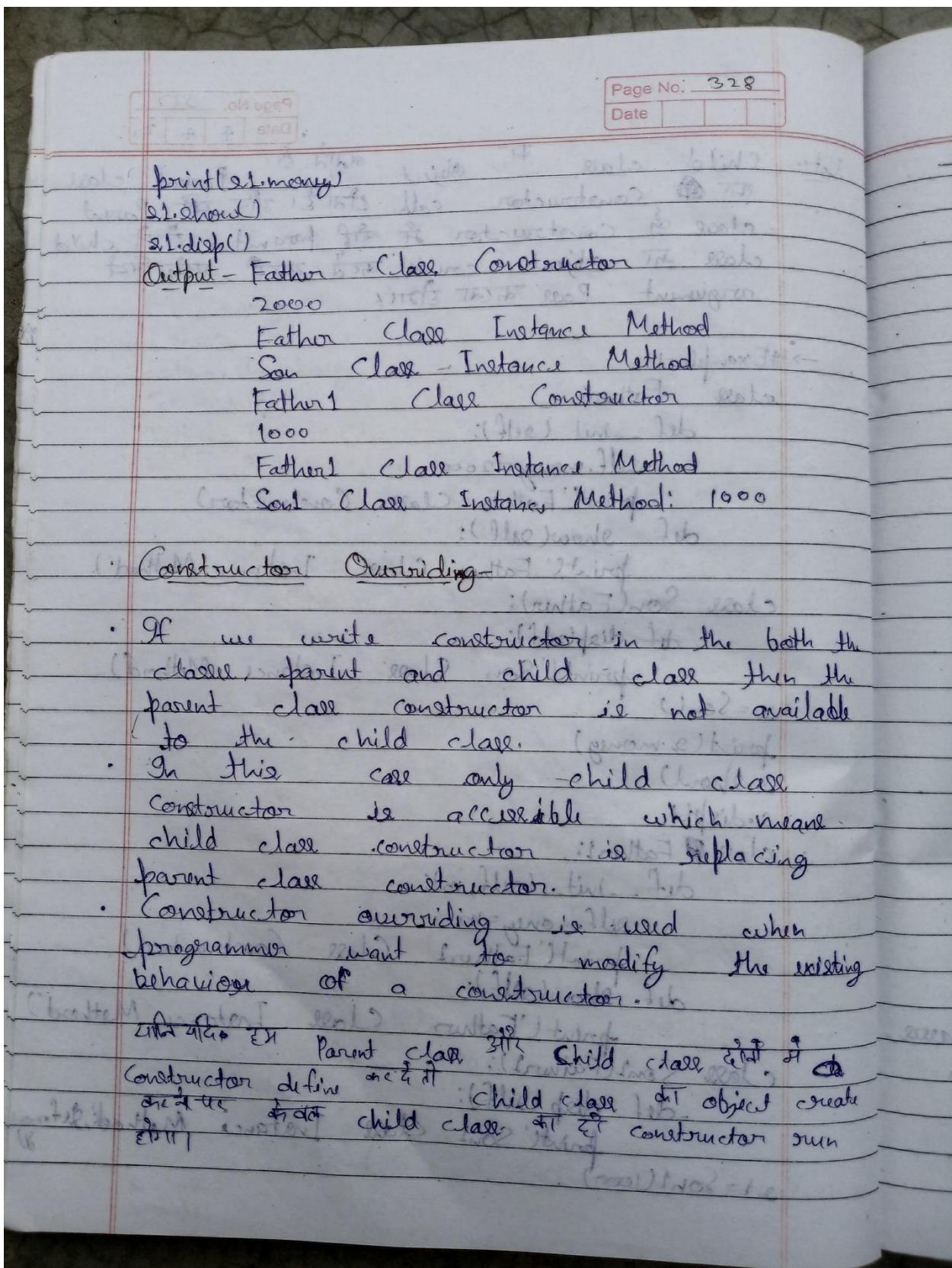
class Son ( Father ):  
 member of class Son

Note - Instance variable\*, <sup>3rd part</sup> start in Method inherit









Page No. 329  
Date

→ # Constructor Overloading Without Parameter

```

class Father: # Parent Class
    def __init__(self):
        self.money = 1000
        print('Father Class Constructor')
    def show(self):
        print('Father Class Instance Method')

```

class Son(Father):

```

    def __init__(self):
        self.money = 5000
        self.car = 'BMW'
    def disp(self):
        print('Son Class Instance Method')

```

s = Son()  
print(s.money)  
print(s.car)  
s.disp()  
s.show()  
f = Father()  
print(f.money)

# Constructor Overloading With Parameter

```

class Father: # Parent Class
    def __init__(self, m):
        self.money = m
        print('Father Class Constructor')
    def show(self):
        print('Father Class Instance Method')

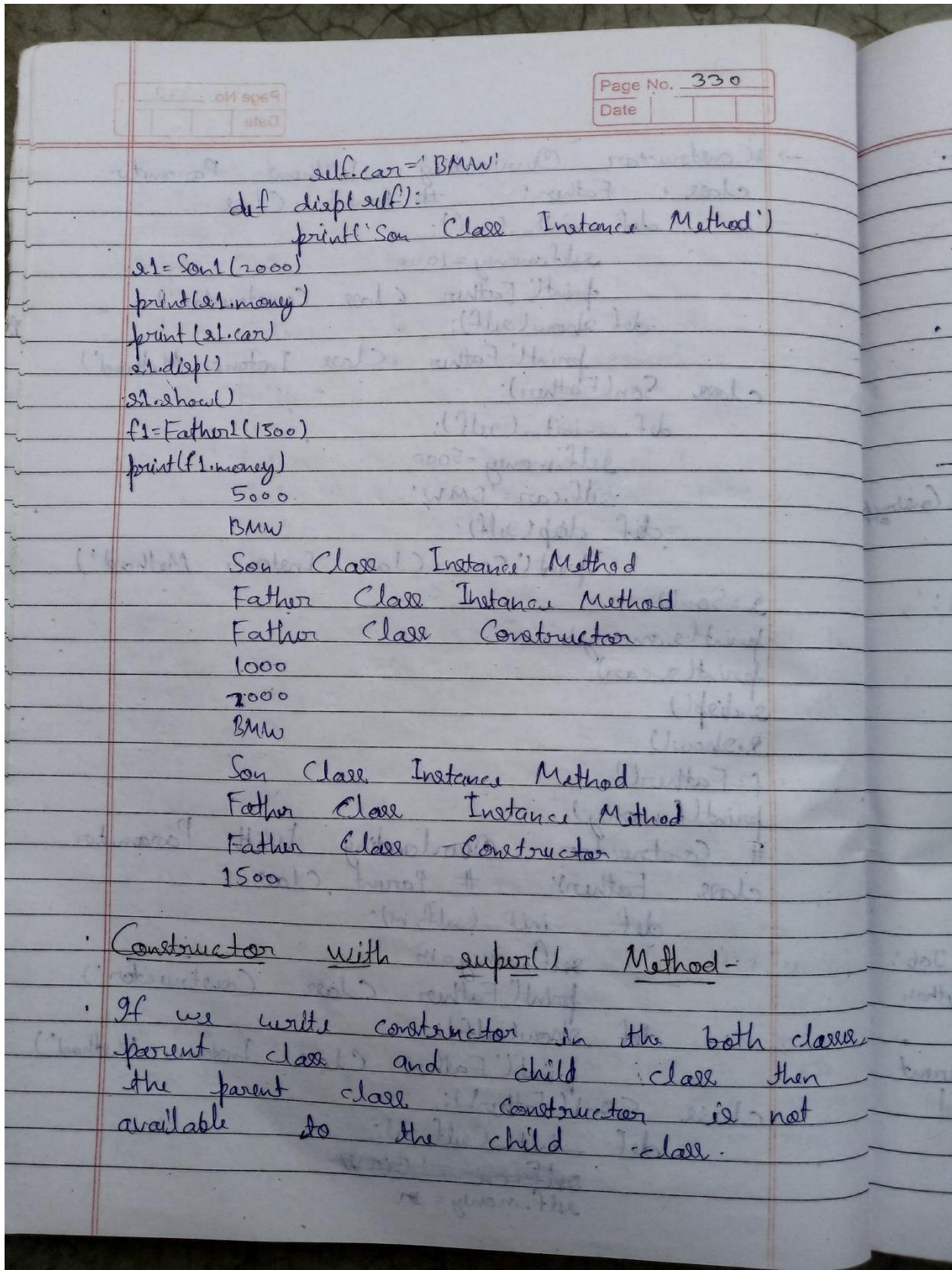
```

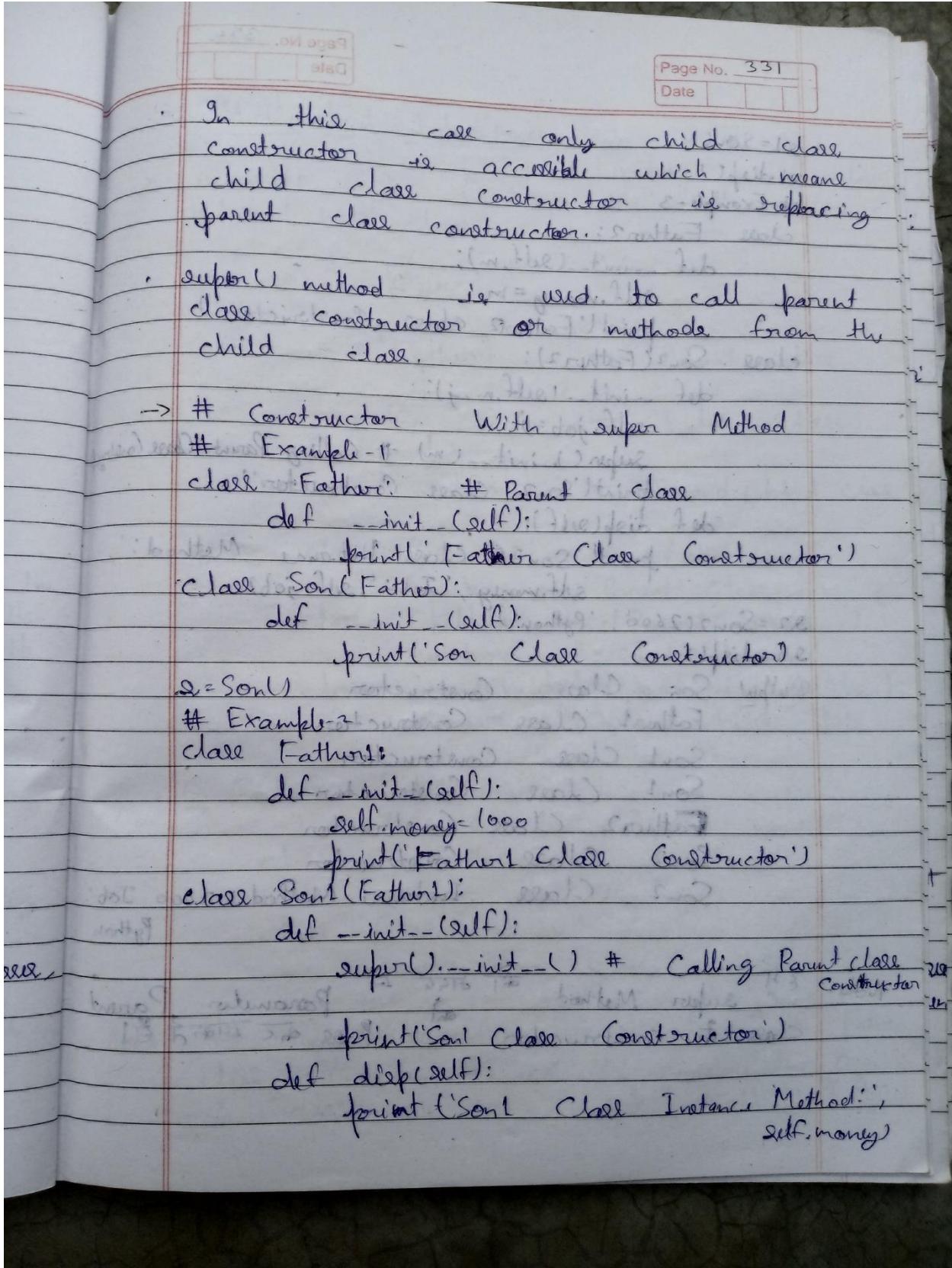
class Son1(Father):

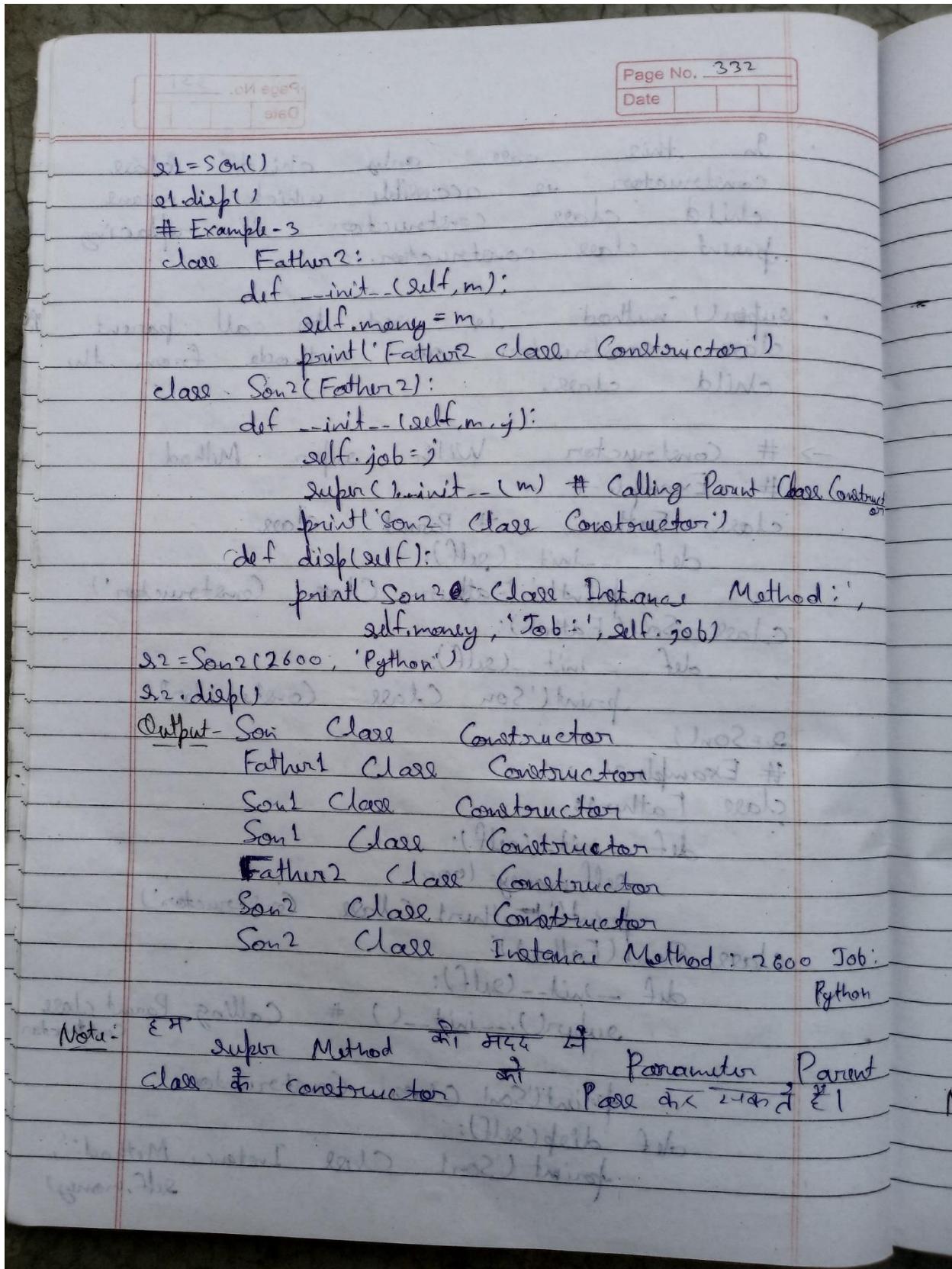
```

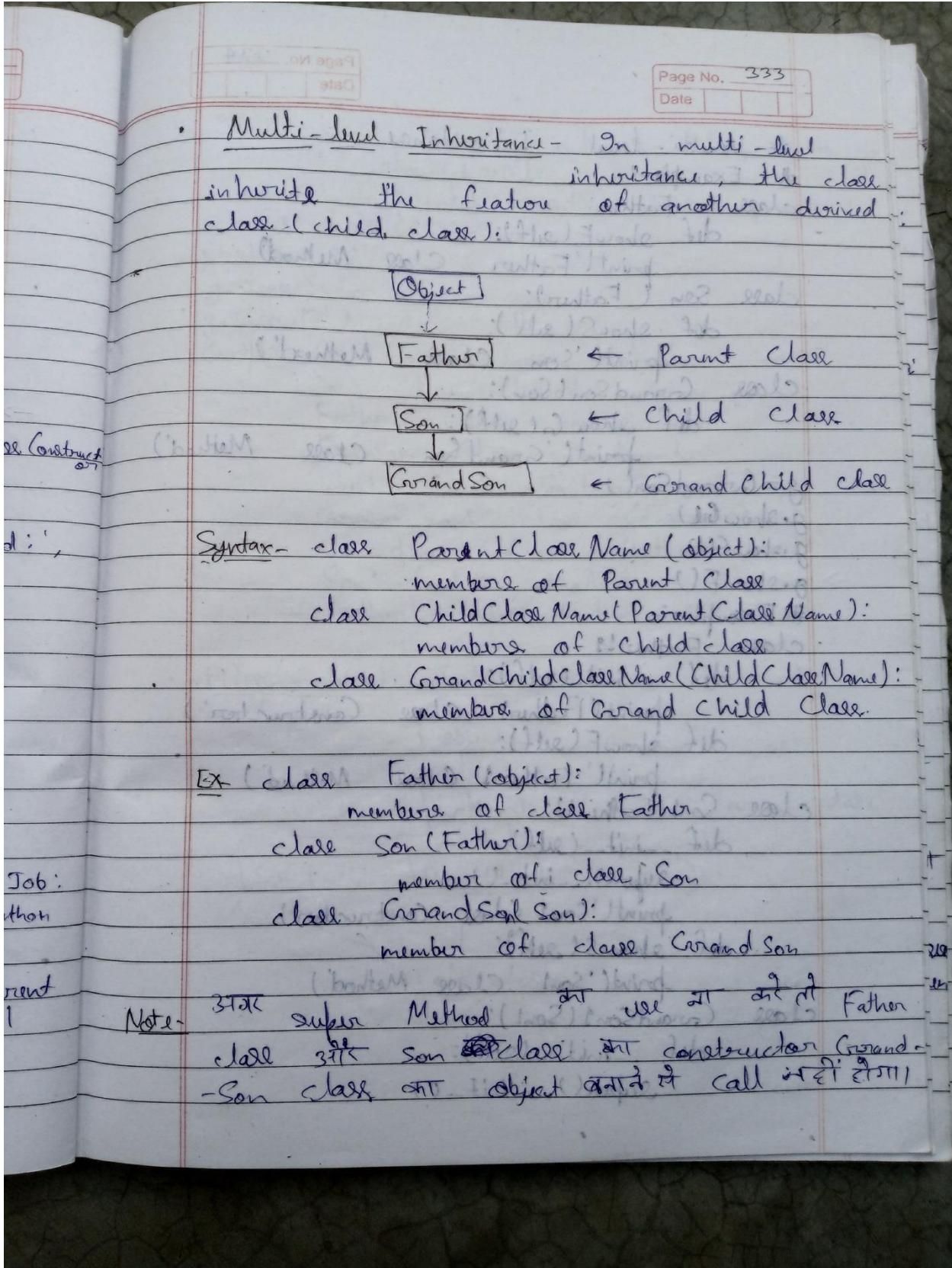
    def __init__(self, m, s):
        self.money = m
        self.car = 'BMW'
        print('Son1 Class Constructor')
    def disp(self):
        print('Son1 Class Instance Method')

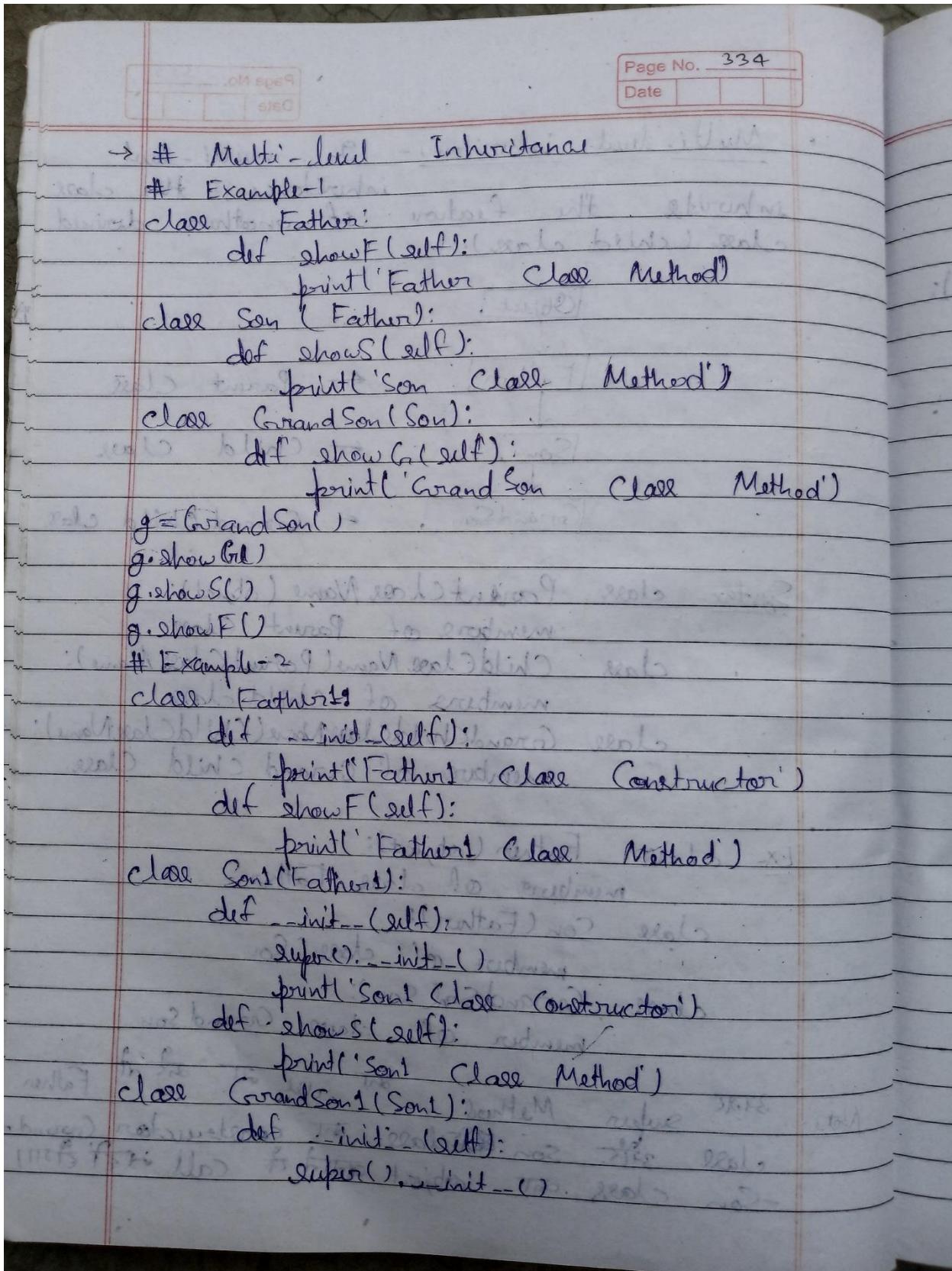
```

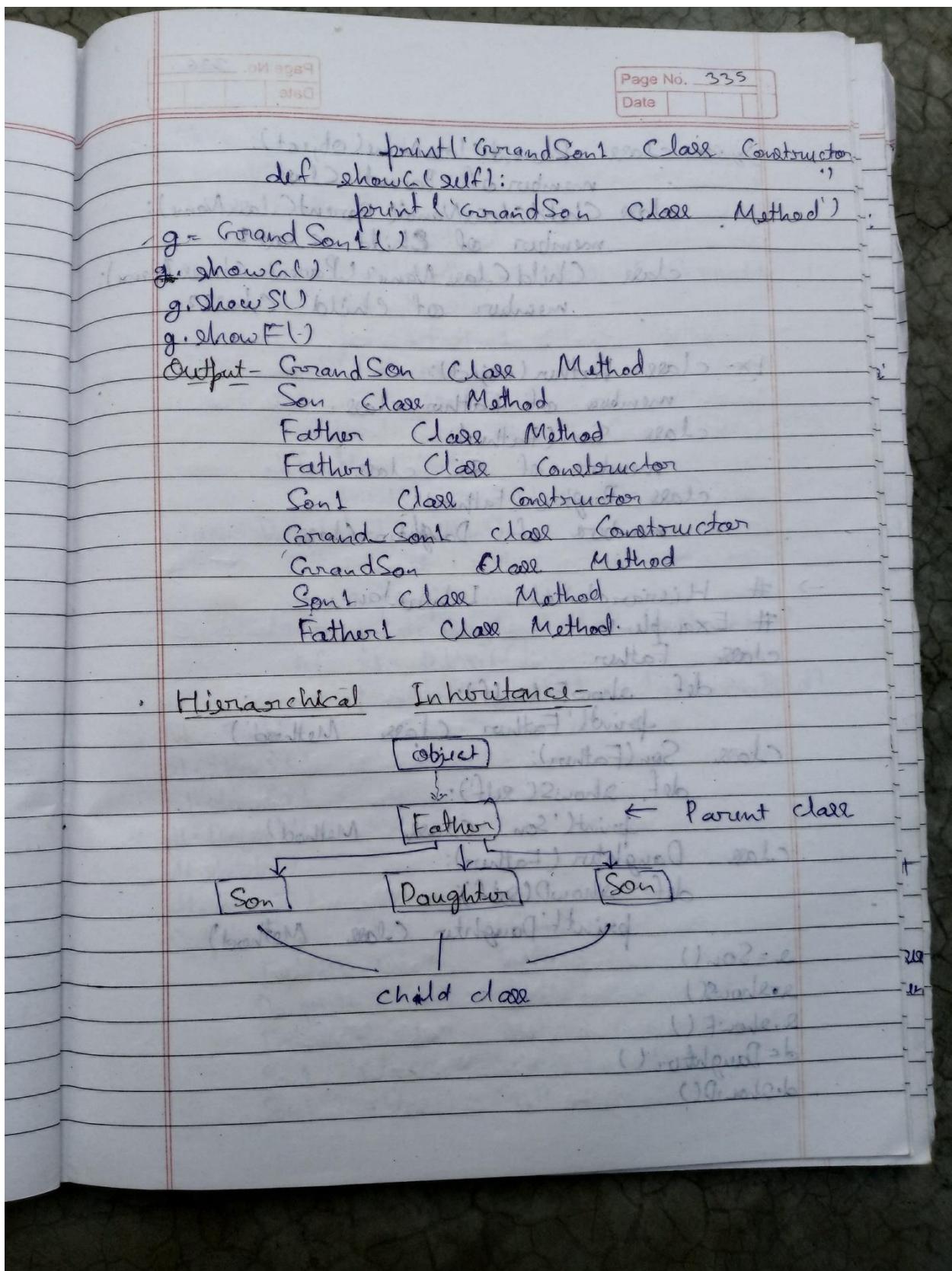


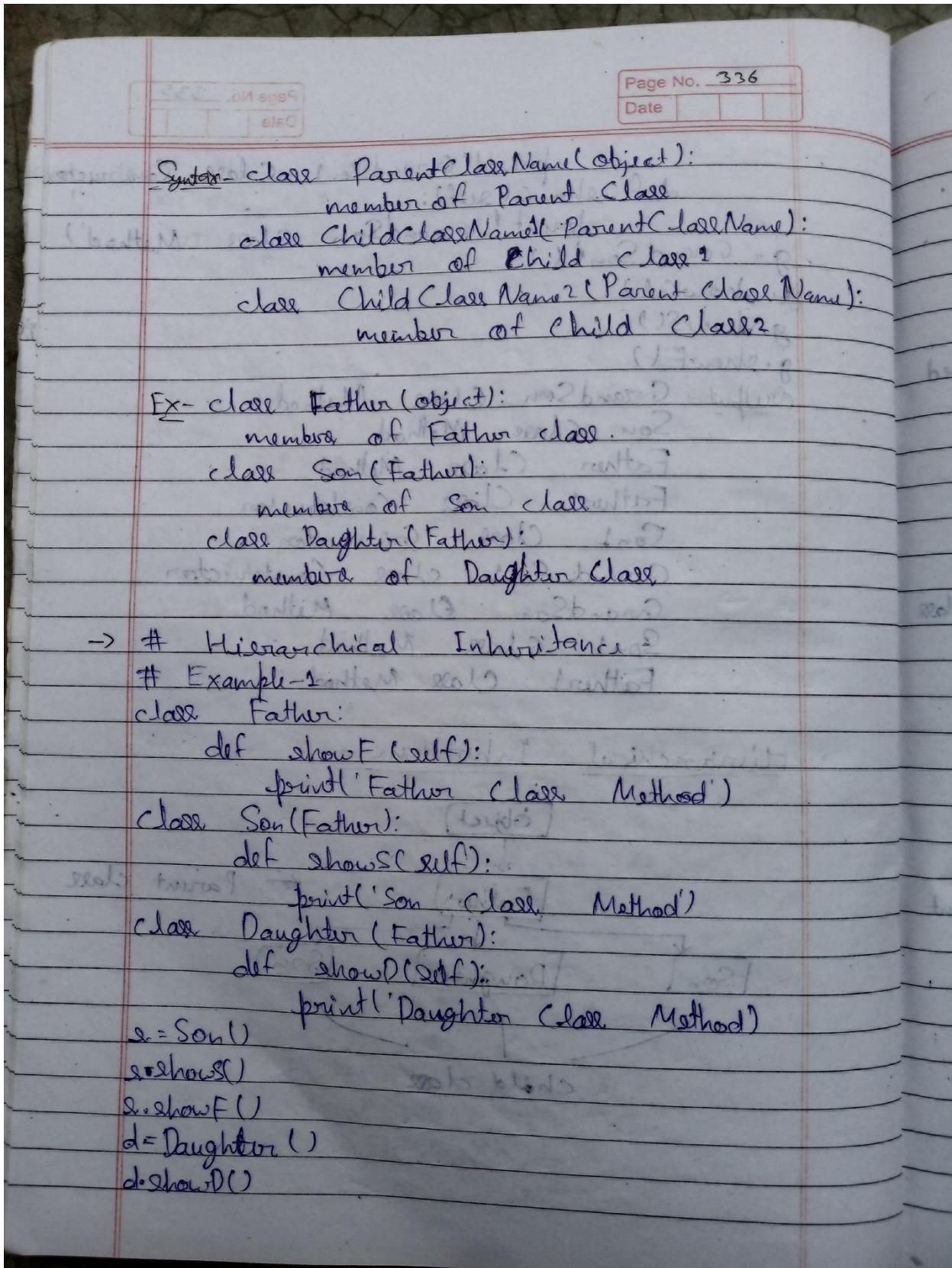


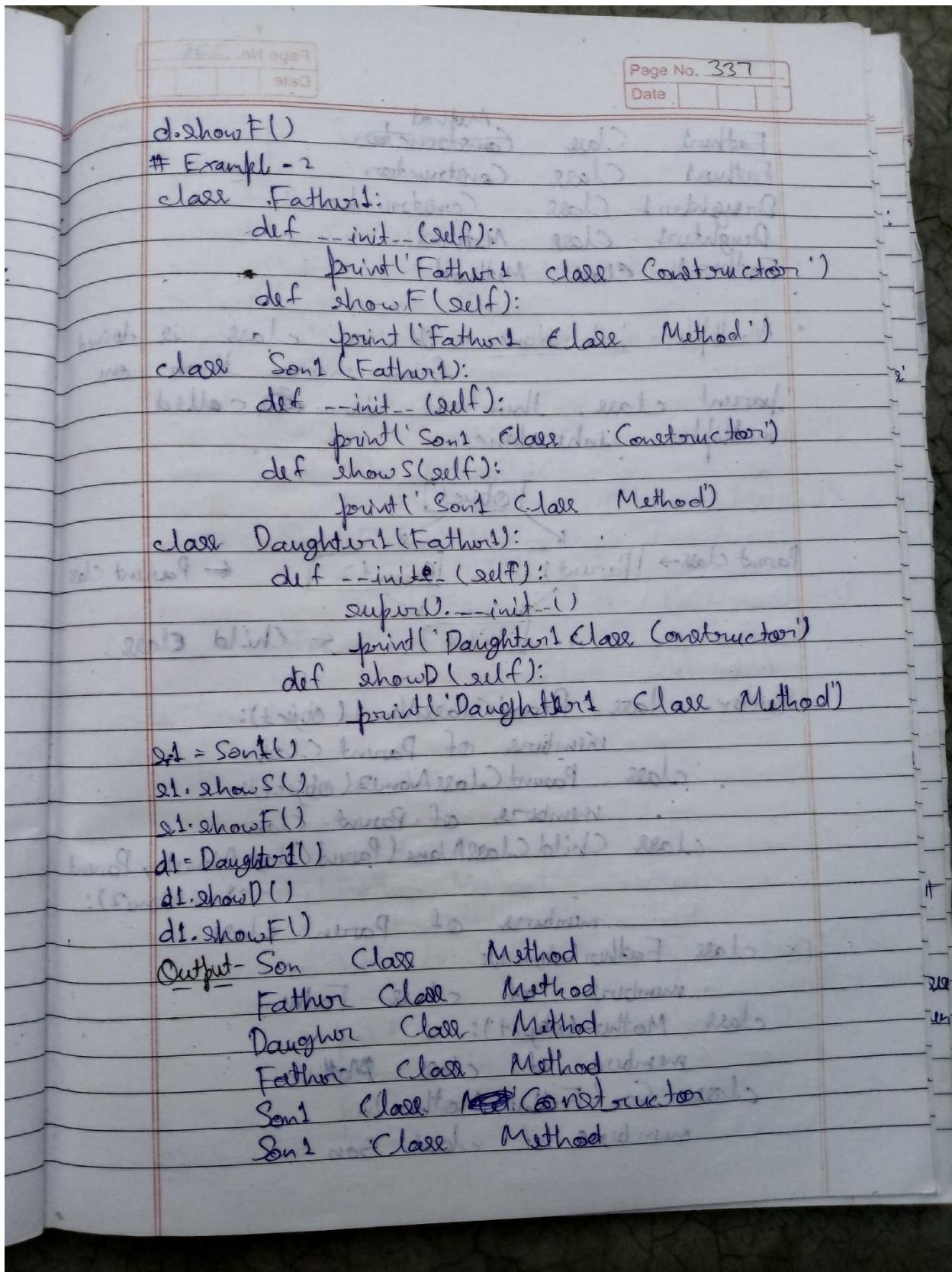


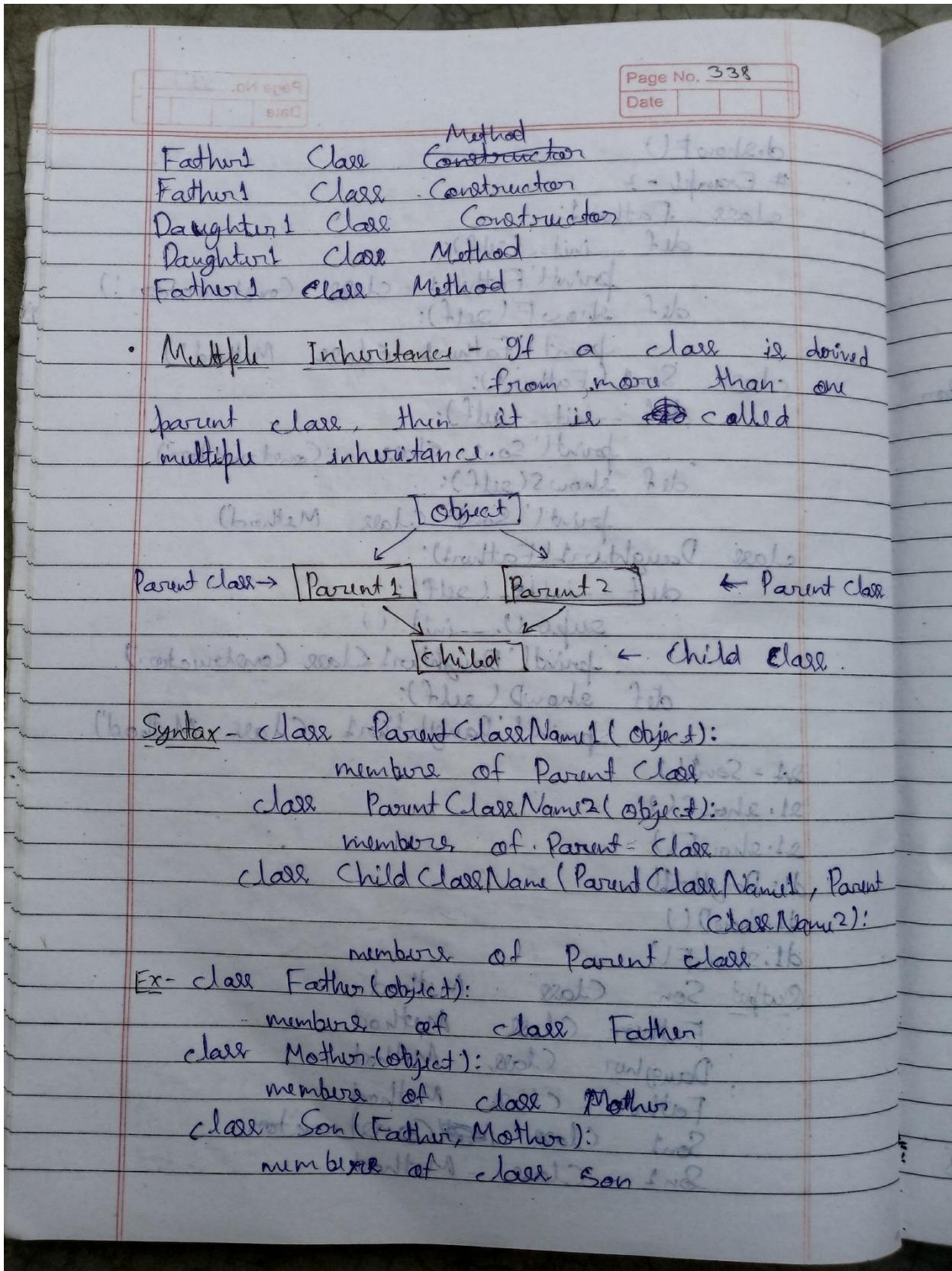


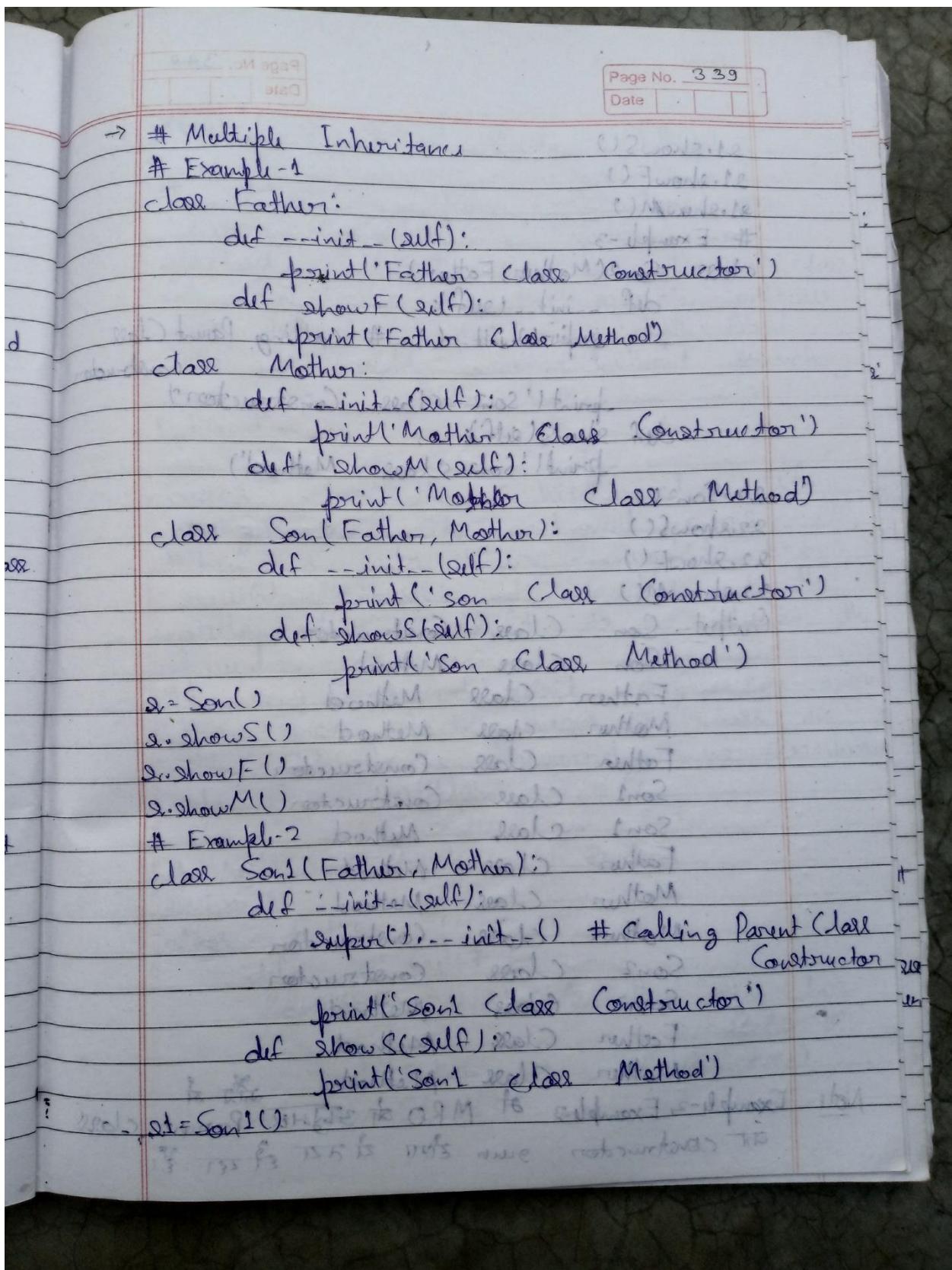


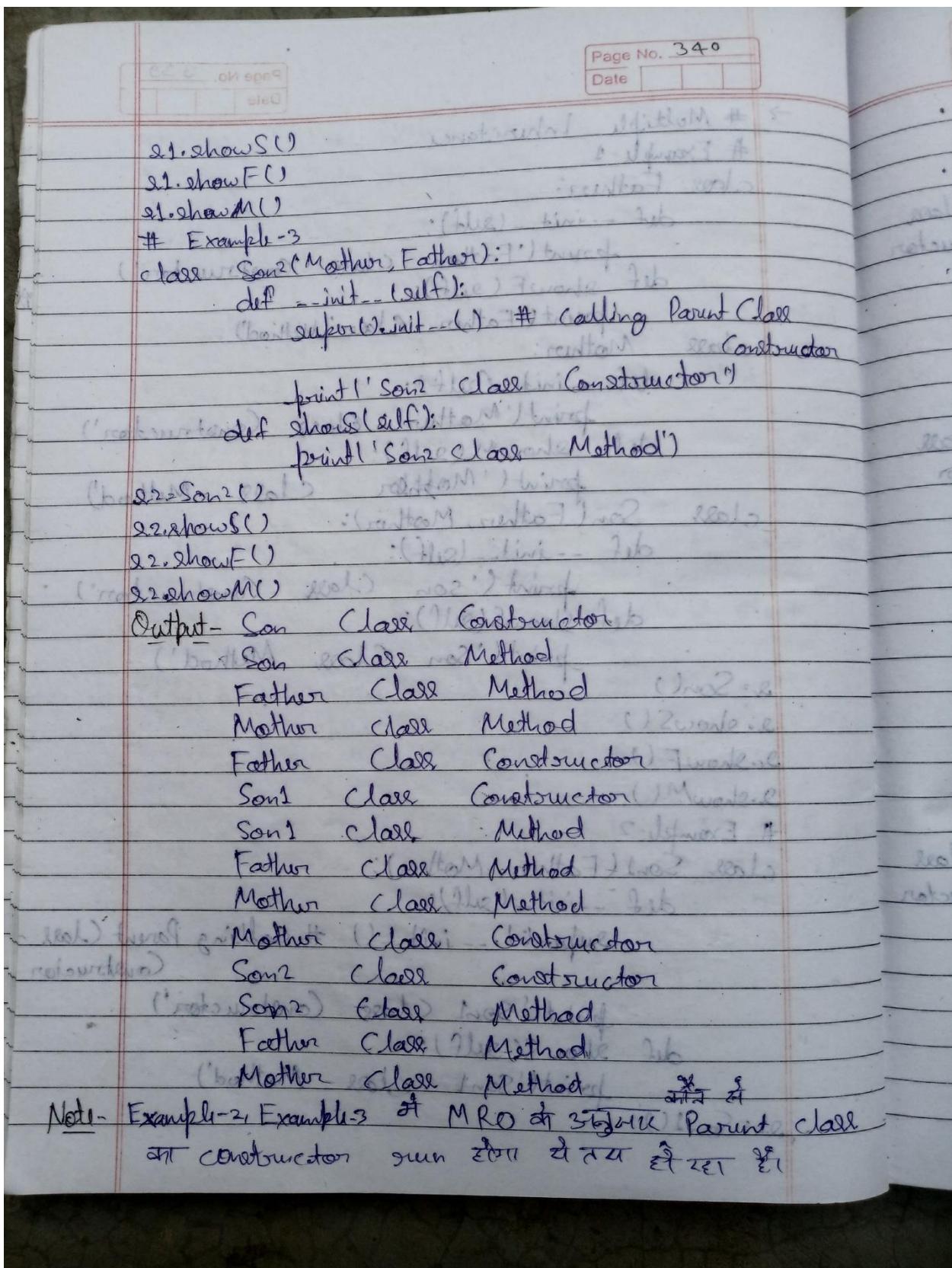


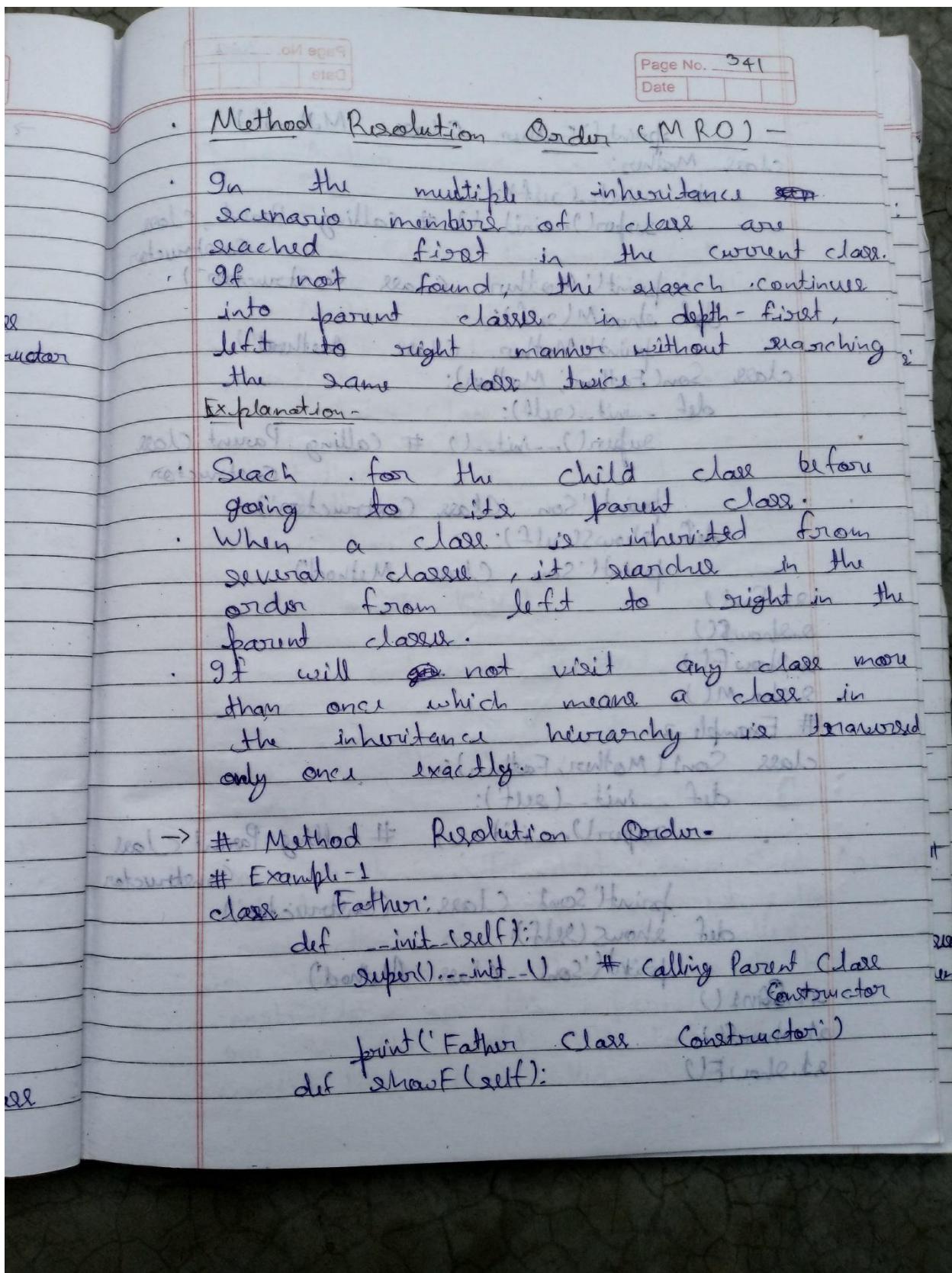


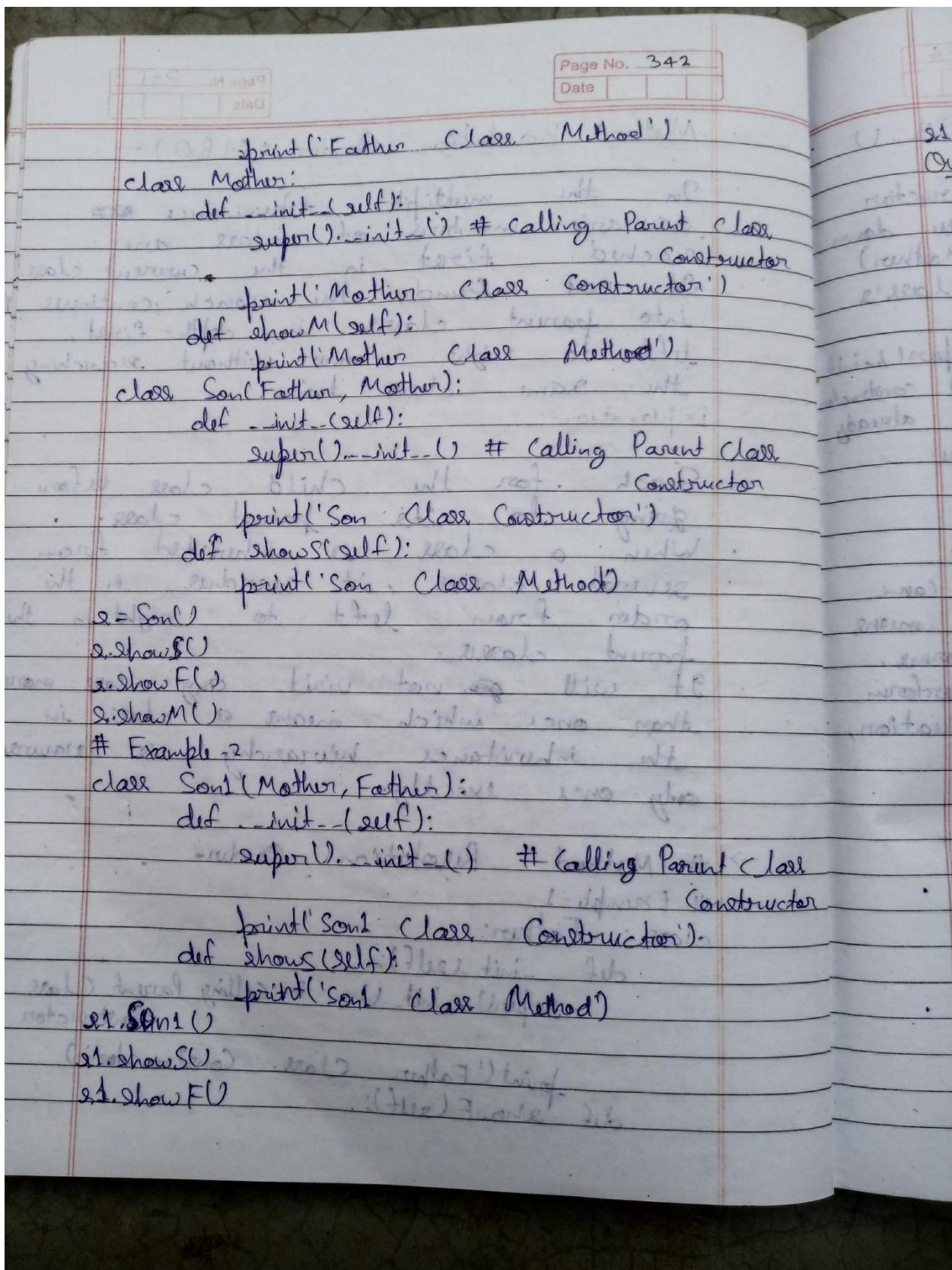












Page No. 343  
Date

s1.showM()

Output - Mother Class Constructor  
 Father Class Constructor  
 Son Class Constructor  
 Son Class ~~Constructor~~ Method  
 Father Class ~~Constructor~~ Method  
 Mother Class Method  
 Father Class Constructor  
 Mother Class Constructor  
 Son Class Constructor  
 Son Class Method  
 Father Class Method  
 Mother Class Method

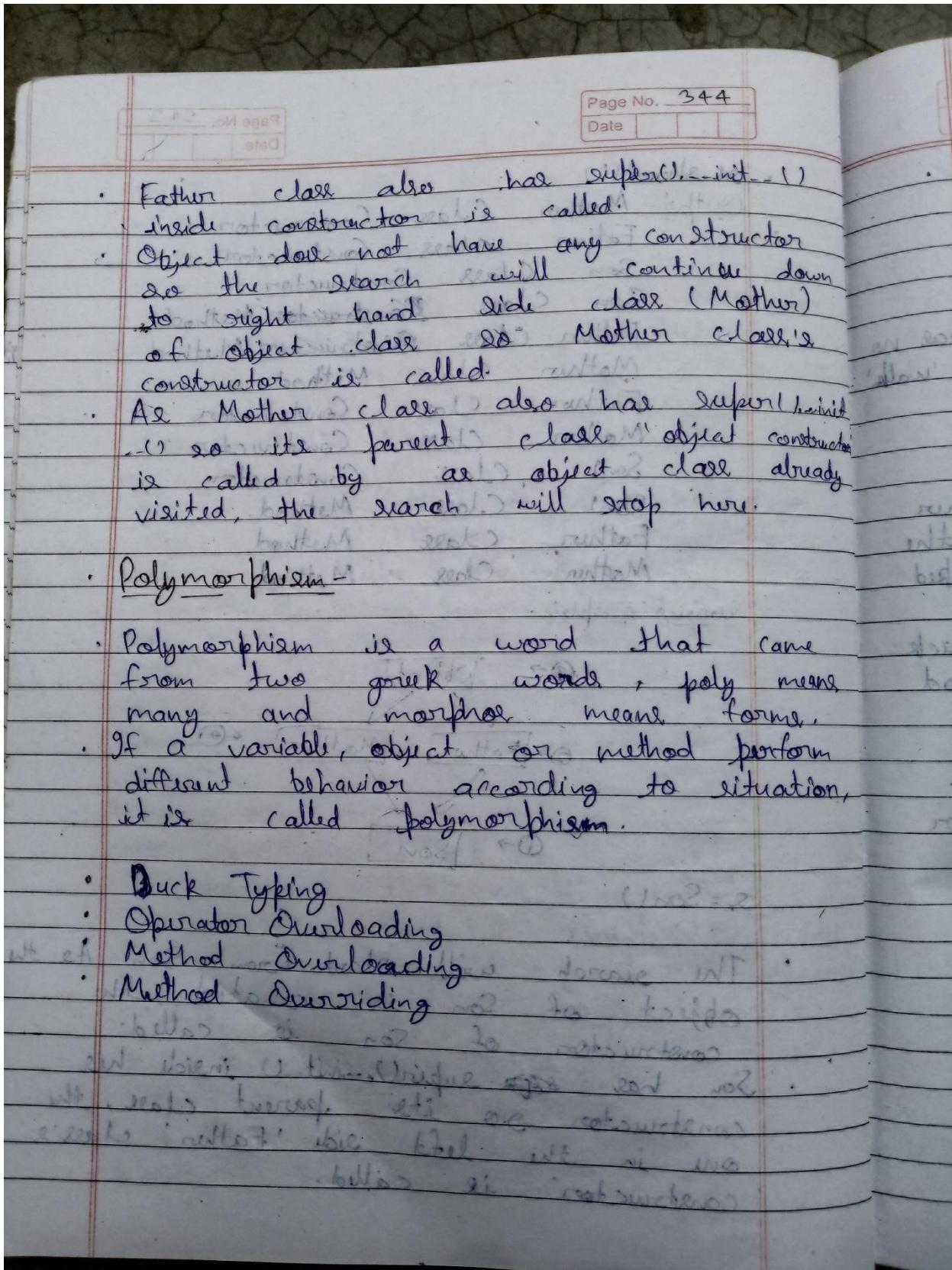
working Explain -

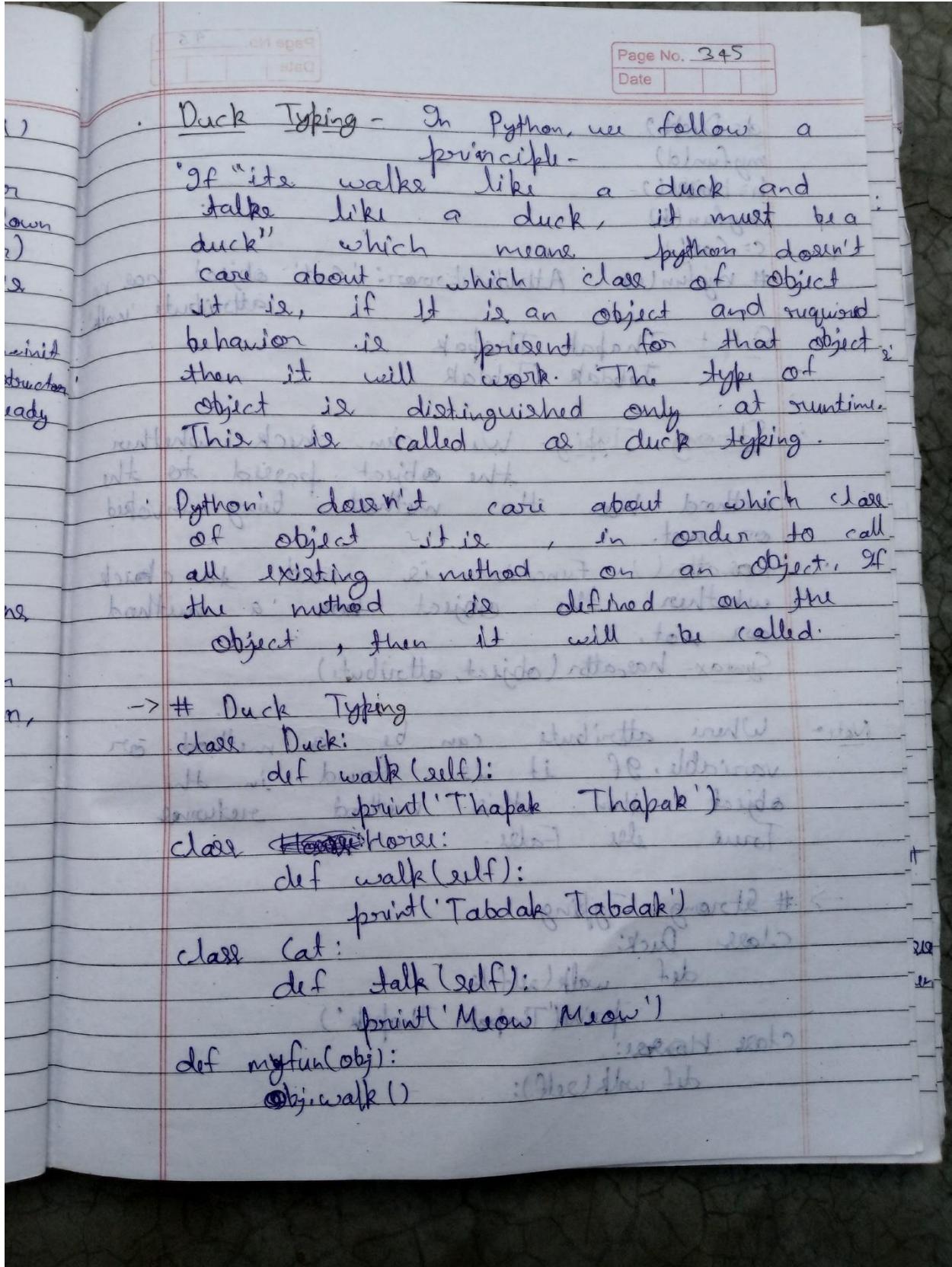
```

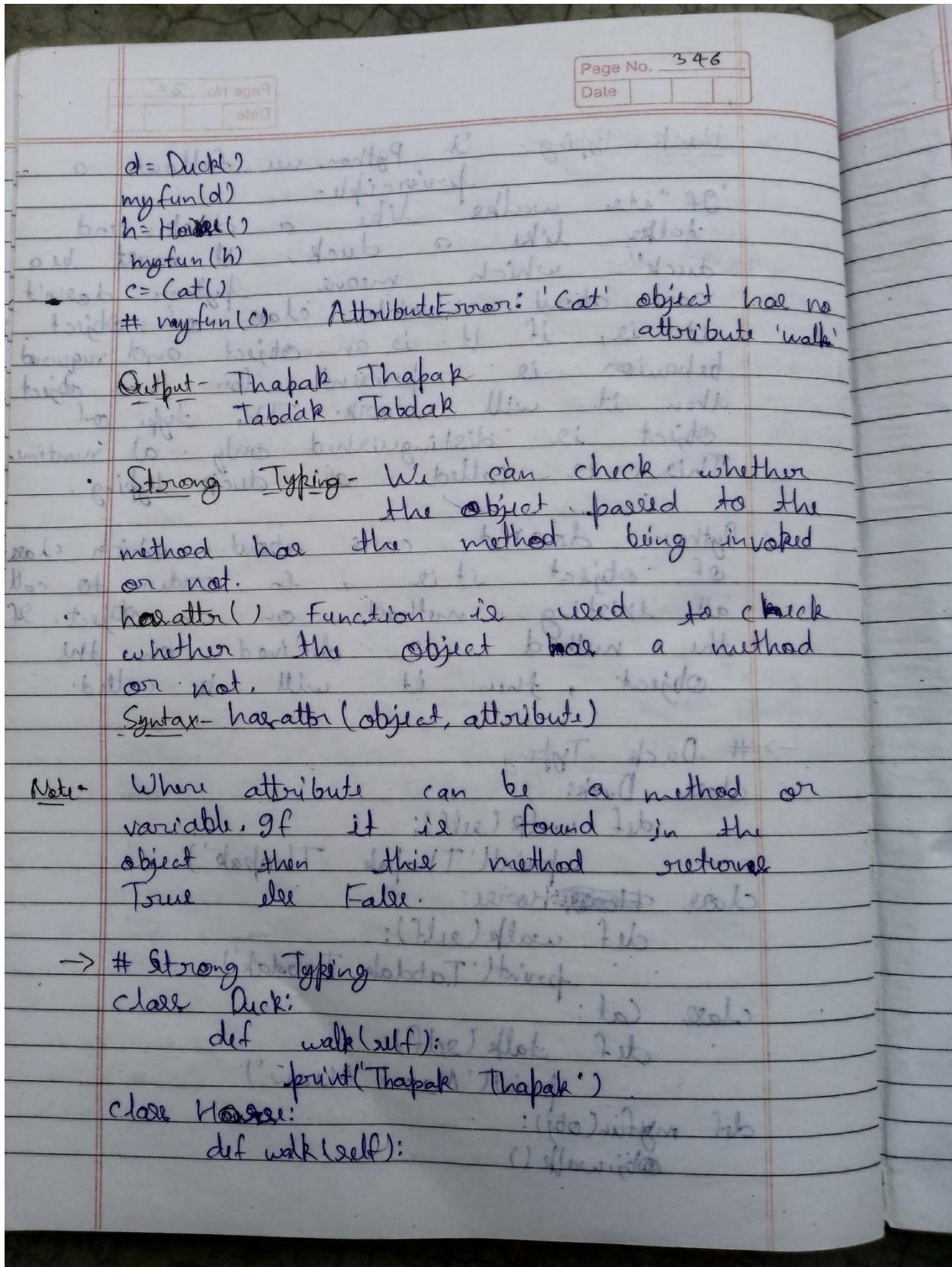
graph TD
    Object[Object] -- ③ --> Father[Father]
    Object[Object] -- ③ --> Mother[Mother]
    Father -- ② --> Son[Son]
    Mother -- ④ --> Son[Son]
    Object[Object] -- ① --> Son[Son]
  
```

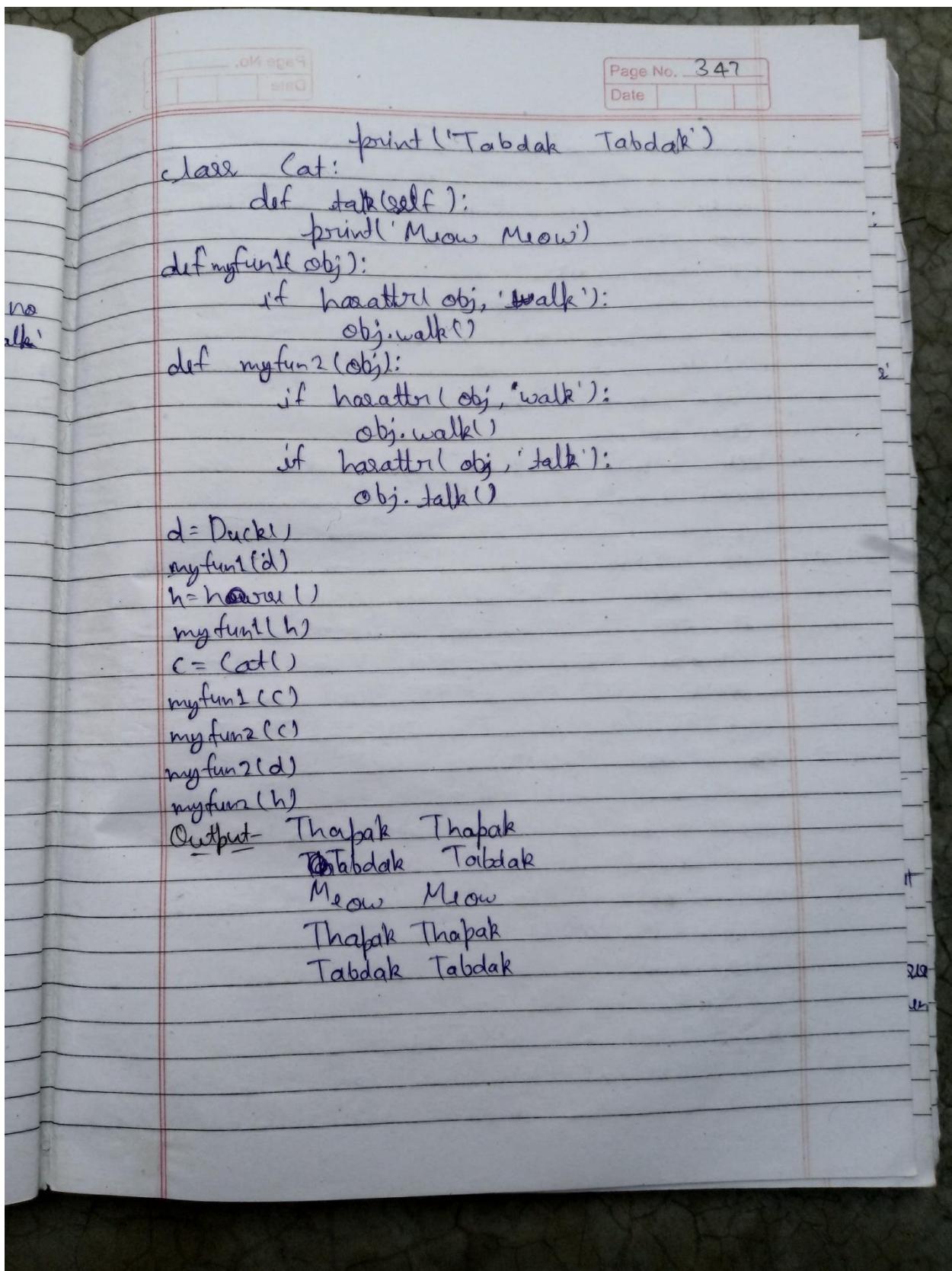
`s = Son()`

- The search will start from the Son. As the object of Son is created, the constructor of Son is called.
- Son has ~~super().\_\_init\_\_()~~ inside his constructor so its parent class, the one in the left side 'Father' class's constructor is called.









Page No. 348  
Date

- Method Overloading - "Python Method overloading जैसा है"
- When more than one method with the same name is defined in the same class it is known as method overloading.
- In Python, If a method is written such that it can perform more than one task, it is called method overloading.
- We achieve method overloading by writing same method with several parameters.

Note- In other programming languages like Java, C++ etc. Python के इनके पास class में एक ही नाम का दो Method दिये गये हैं। इसका prototype अलग-अलग हमेशा last के दिये गये हैं। इसका function का नाम इसकी।

→ ~~class~~ class MyClass:

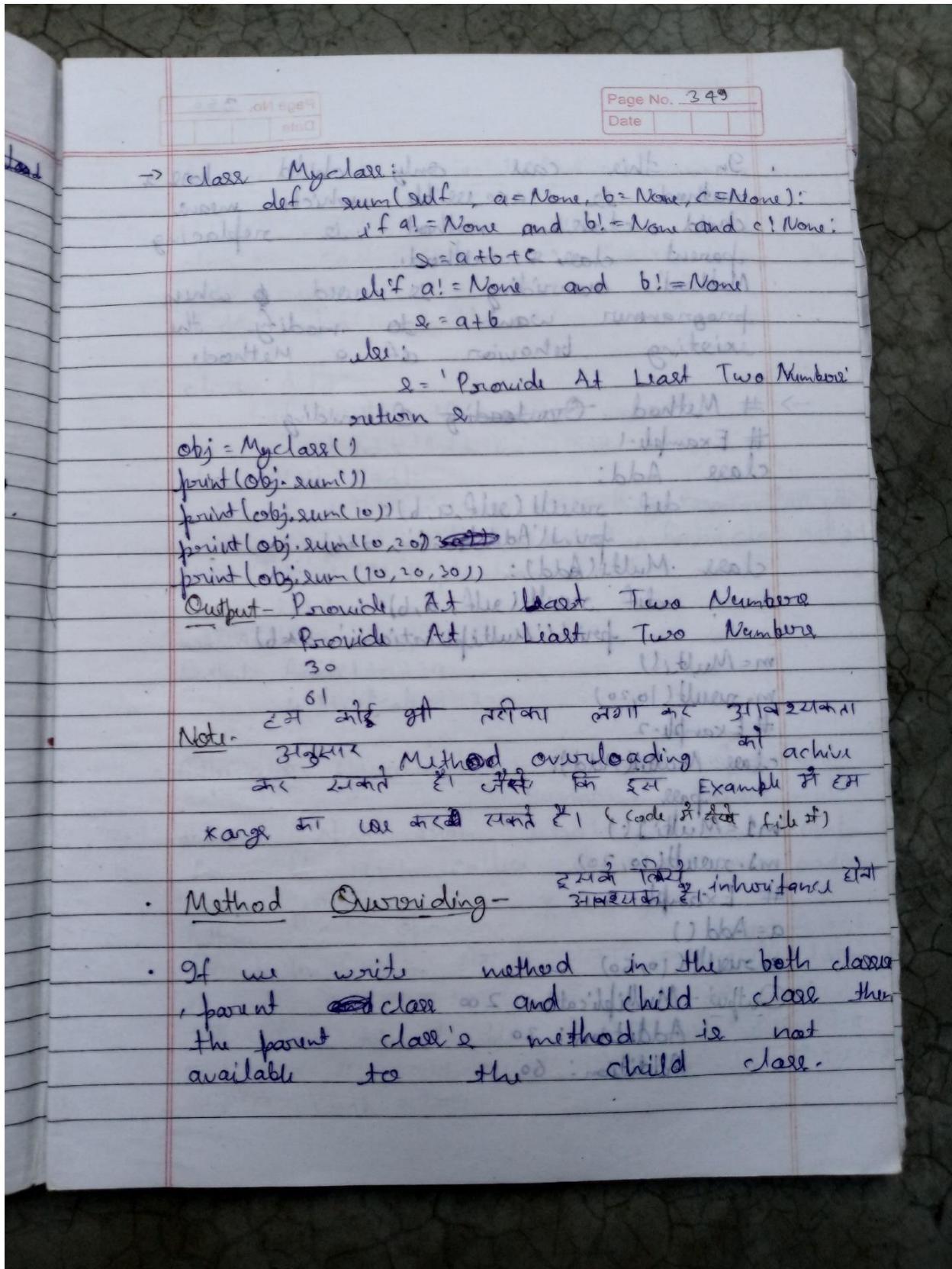
```

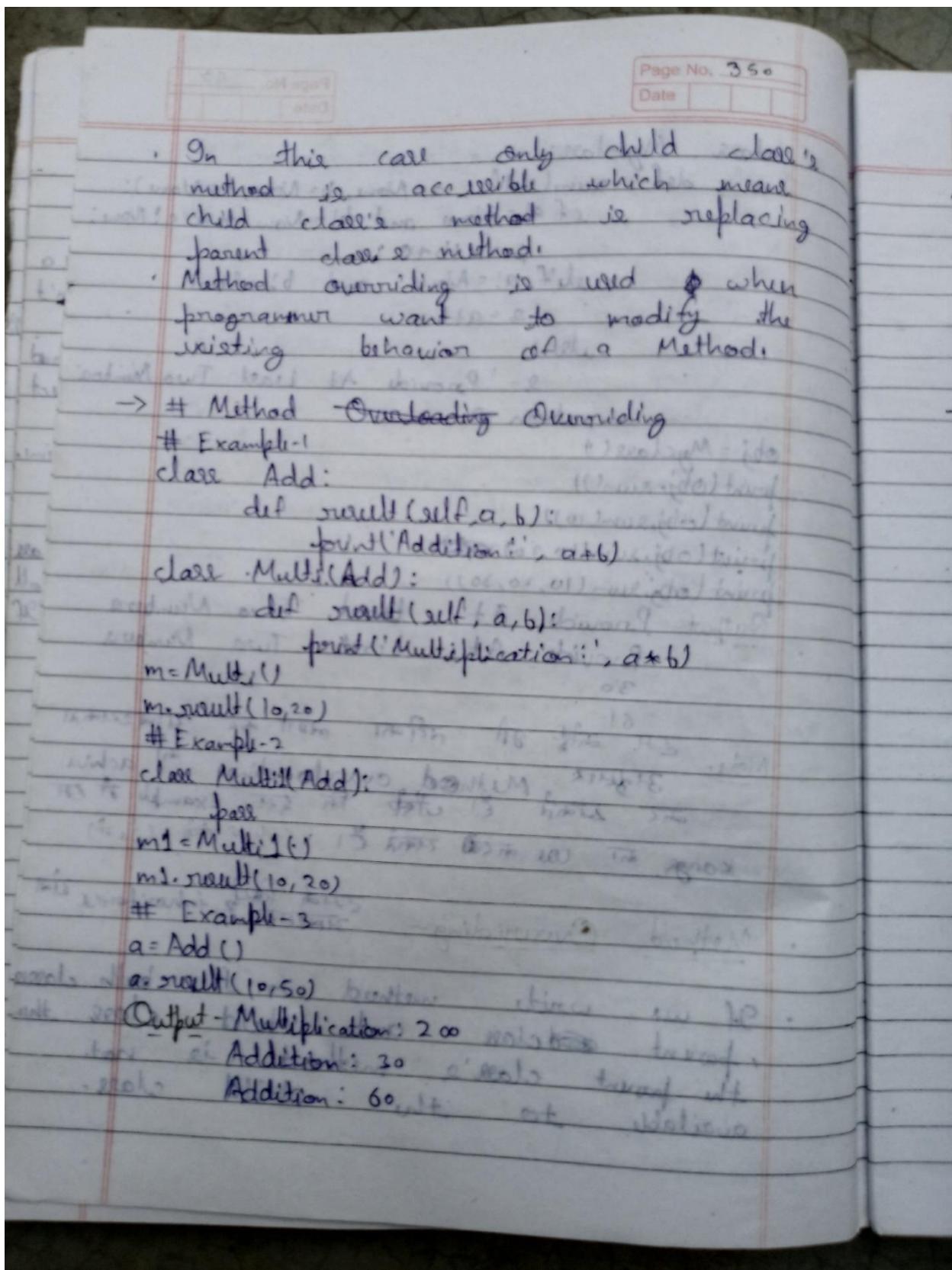
def sum(self, a):
    print("1st Sum Method", a)
def sum(self):
    print("2nd Sum Method")

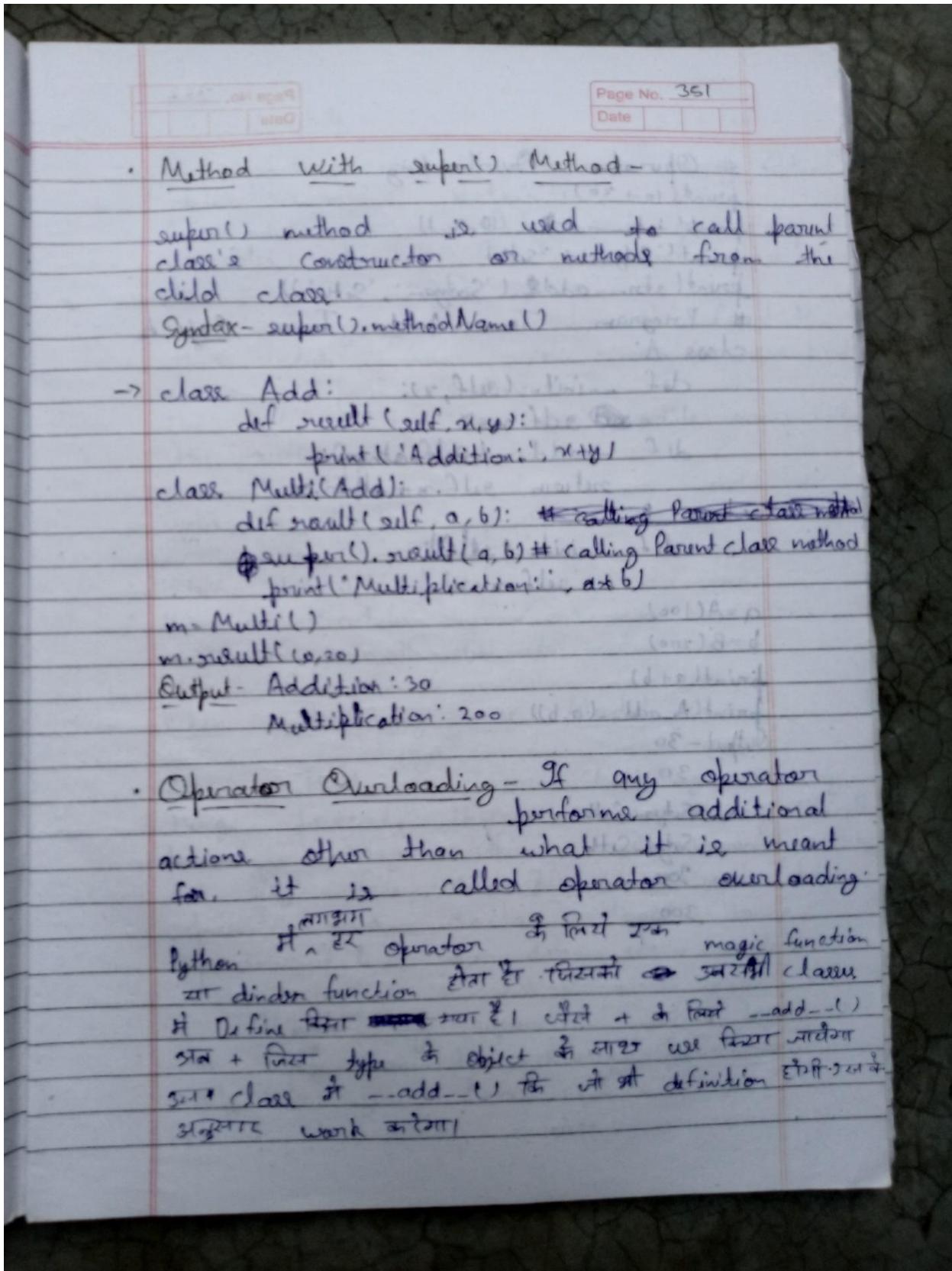
```

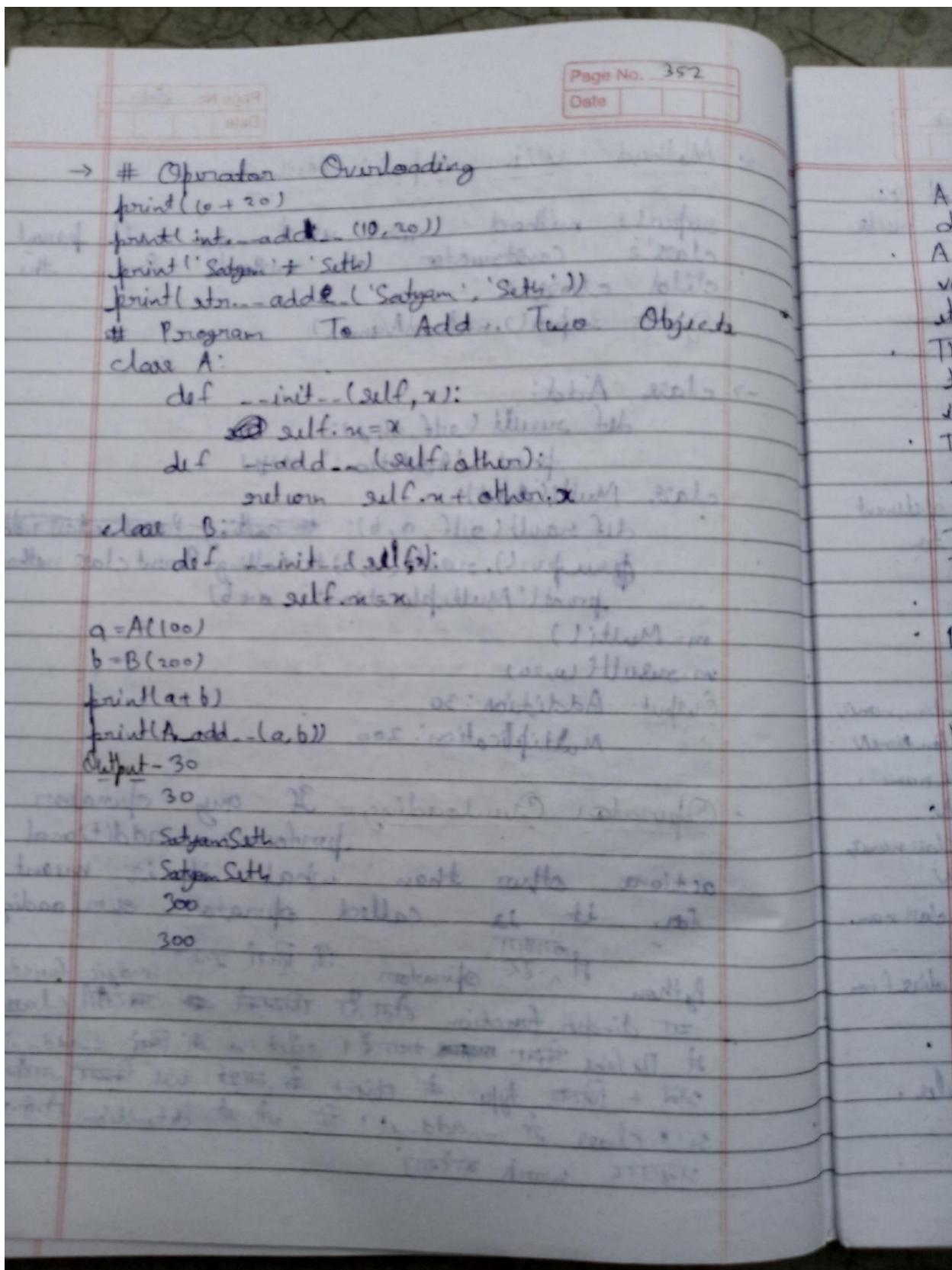
obj = MyClass()  
obj.sum()  
obj.sum(10)

Output- 2nd Sum Method  
TypeError: sum() takes 1 positional argument, but 2 were given









Module

Page No. 353  
Date 5 4 20

- A module is a file containing Python definitions and statements.
- A module is a file containing group of variables, methods, functions and classes etc.
- They are executed only the first time the module name is encountered in an import statement.
- The file name is the module name with the suffix .py appended.

Type of Module -

- User-defined Module
- Built-in Module
  - Ex- array, math, sys etc.

When and Why we use Modules -

- Assume that you are building a very large project, it will be ~~is~~ very difficult to manage all logic within one single file so if you want to separate your similar logic to a separate file, you can use modules.
- It will not only separate your logic but also help you to debug your code easily as you know which logic is defined in which module.

Page No. 354  
Date \_\_\_\_\_

- When a module is developed, it can be reused in any program that needs that module.

Ex-

```

graph LR
    A[geekshows.py] --> B[Database db.py]
    A --> C[Calculation cal.py]
    A --> D[Searching search.py]
  
```

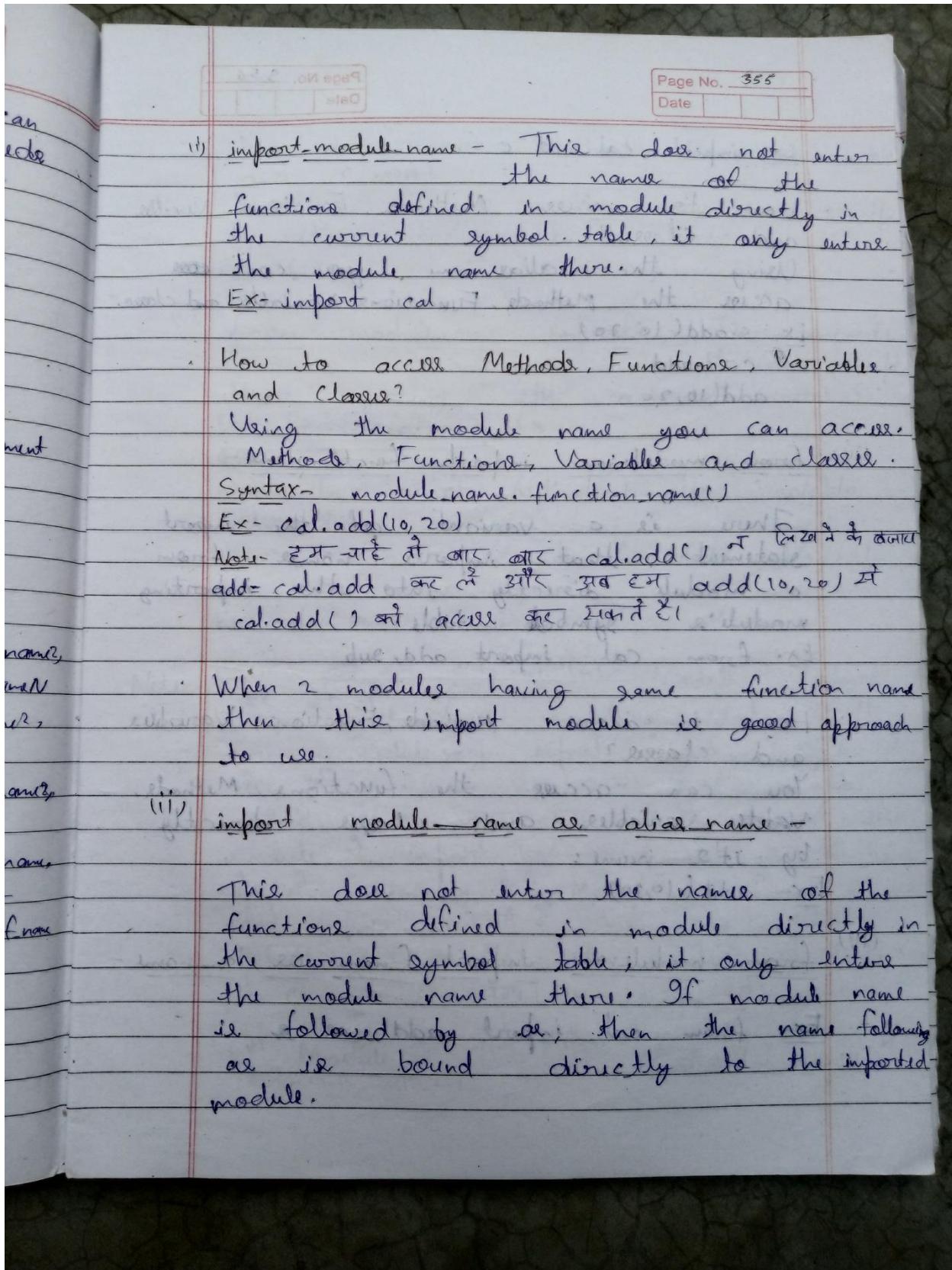
How to use Module - import statement

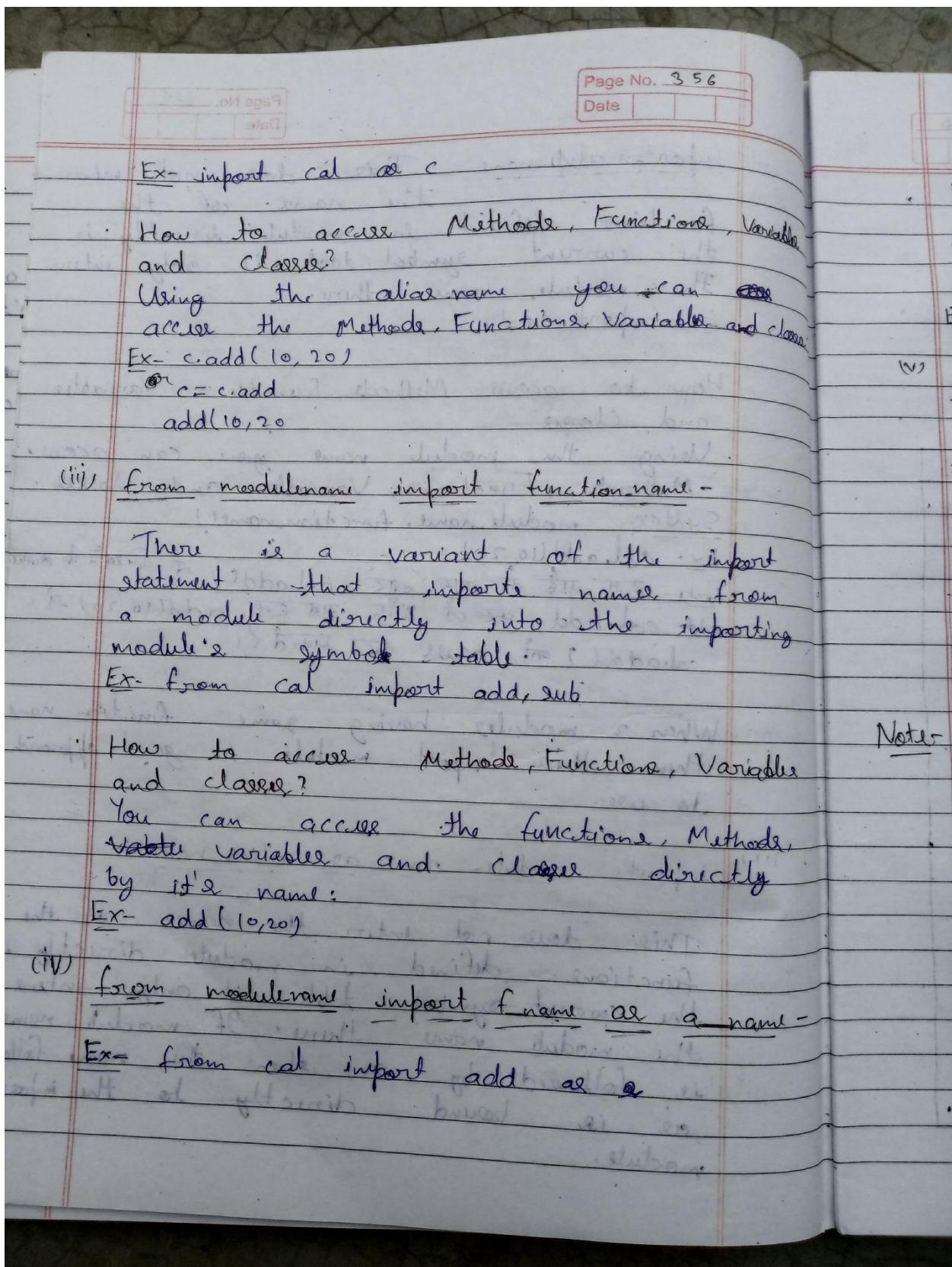
import module-name

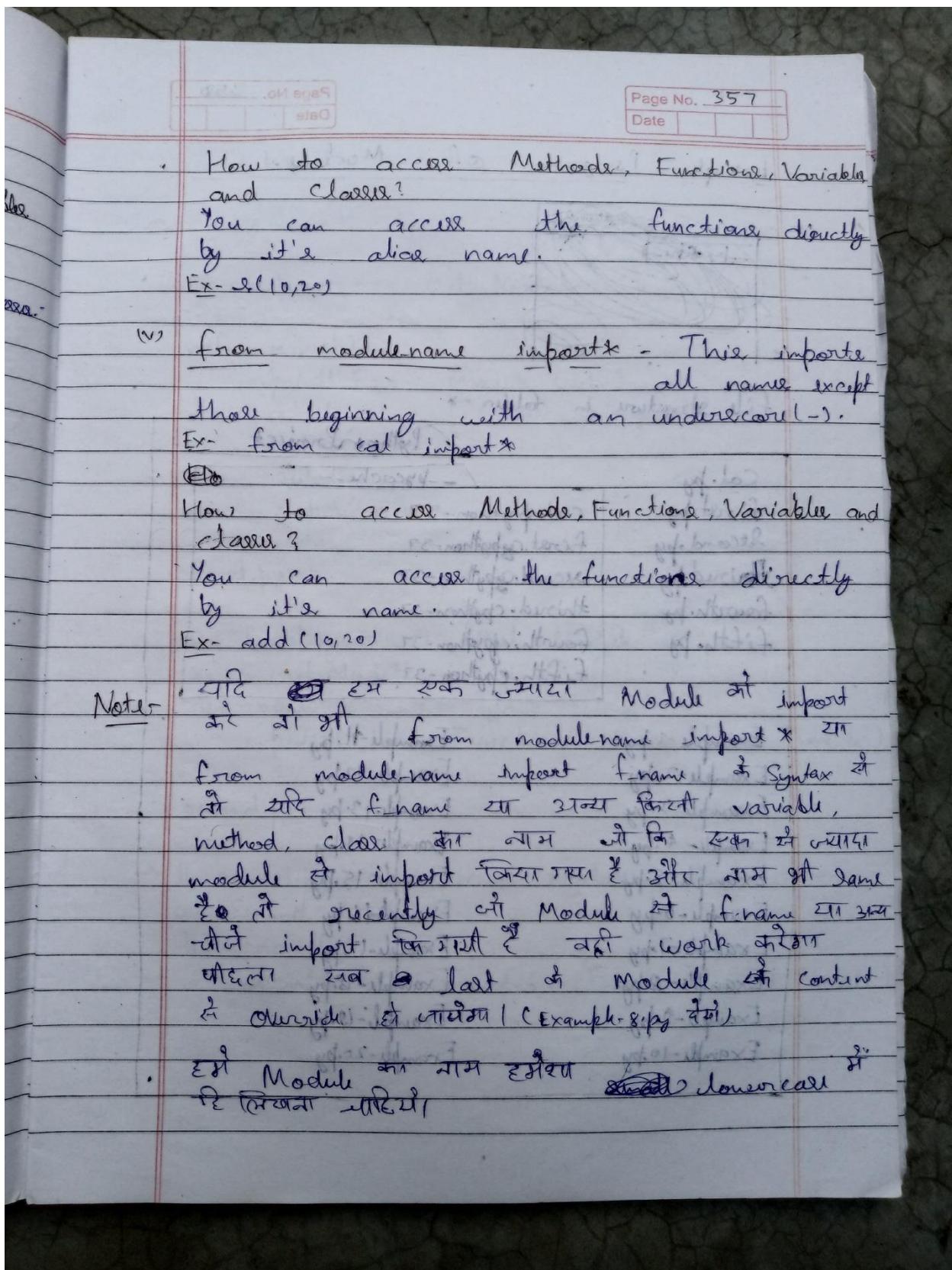
Syntax-

- import module-name
- import module-name as alias-name
- from module-name import function-name1, function-name2, ... , function-nameN
- from module-name import var-name1, variable-name2, ... , varnameN
- from module-name import class-name1, class-name2, ... , class-nameN
- from module-name import varname, fun-name, class-name, method-name
- from module-name import fun-name as alias-fun
- from module-name import \*

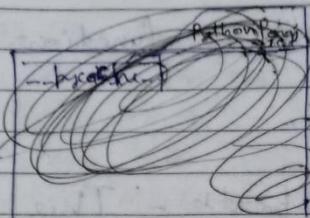
Note - Module can import other modules.



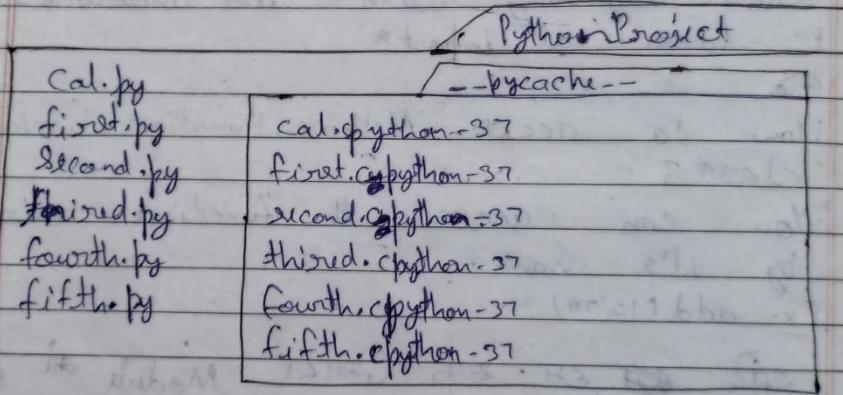




## Example Program of Module



file structure in folder →



## Example -1 .py

## Example-2.py

Example-3. by

Example - 4. by

Example - 5. by

Example - 6. by

Example - 3. py

- example - copy

Example-8.py  
Ex-11.91

Example-9.py

Example-10. by

卷之三

— 1 —

### Example -11 . py

### Example - 12.12

### Example-13.

### Example 1 top

Example- 15. by

### Example 16: by

Example- $\text{C}_2\text{H}_4$

$f_{\text{sample}} = 18 \text{ Hz}$

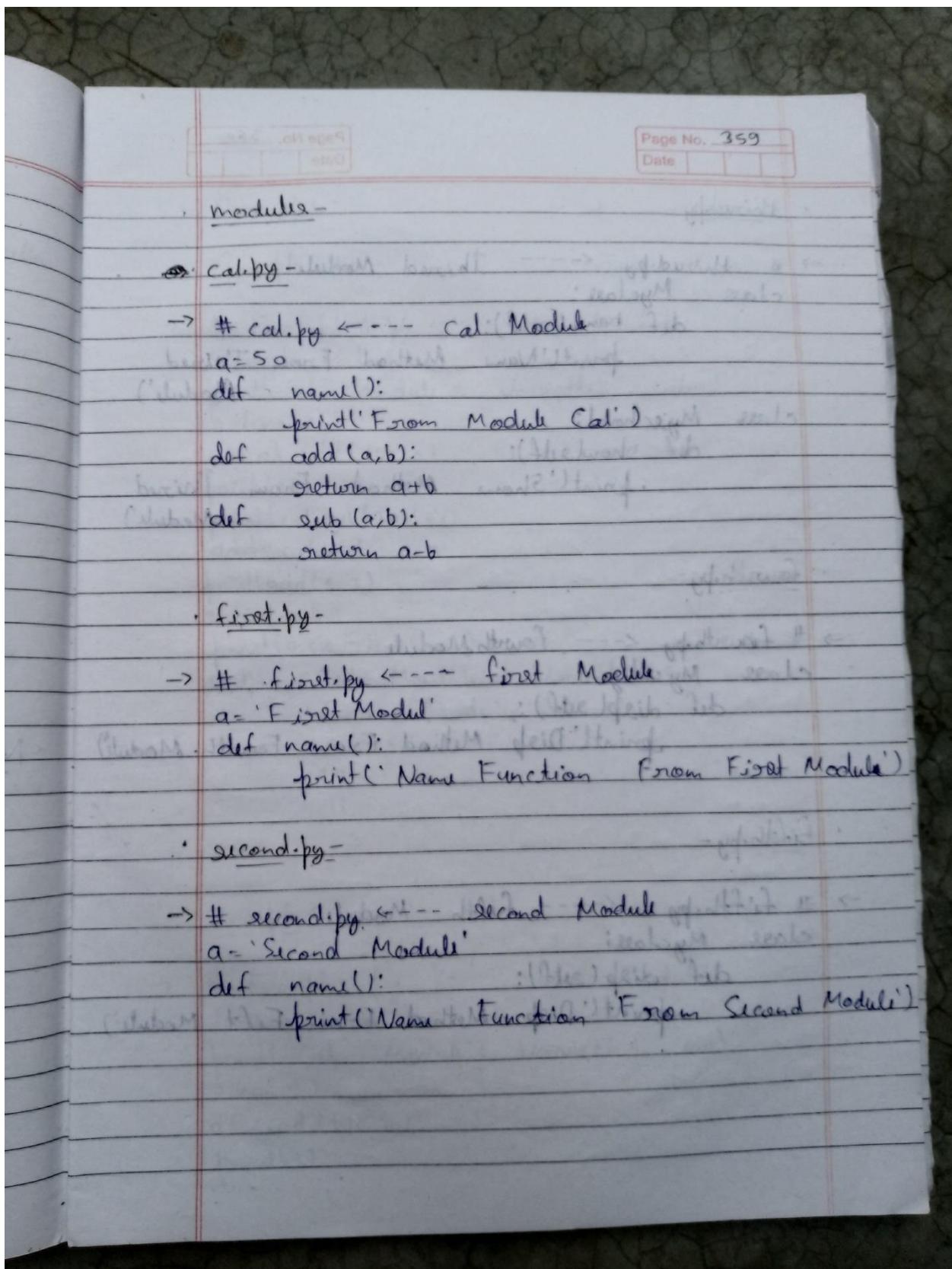
Example - 19, py

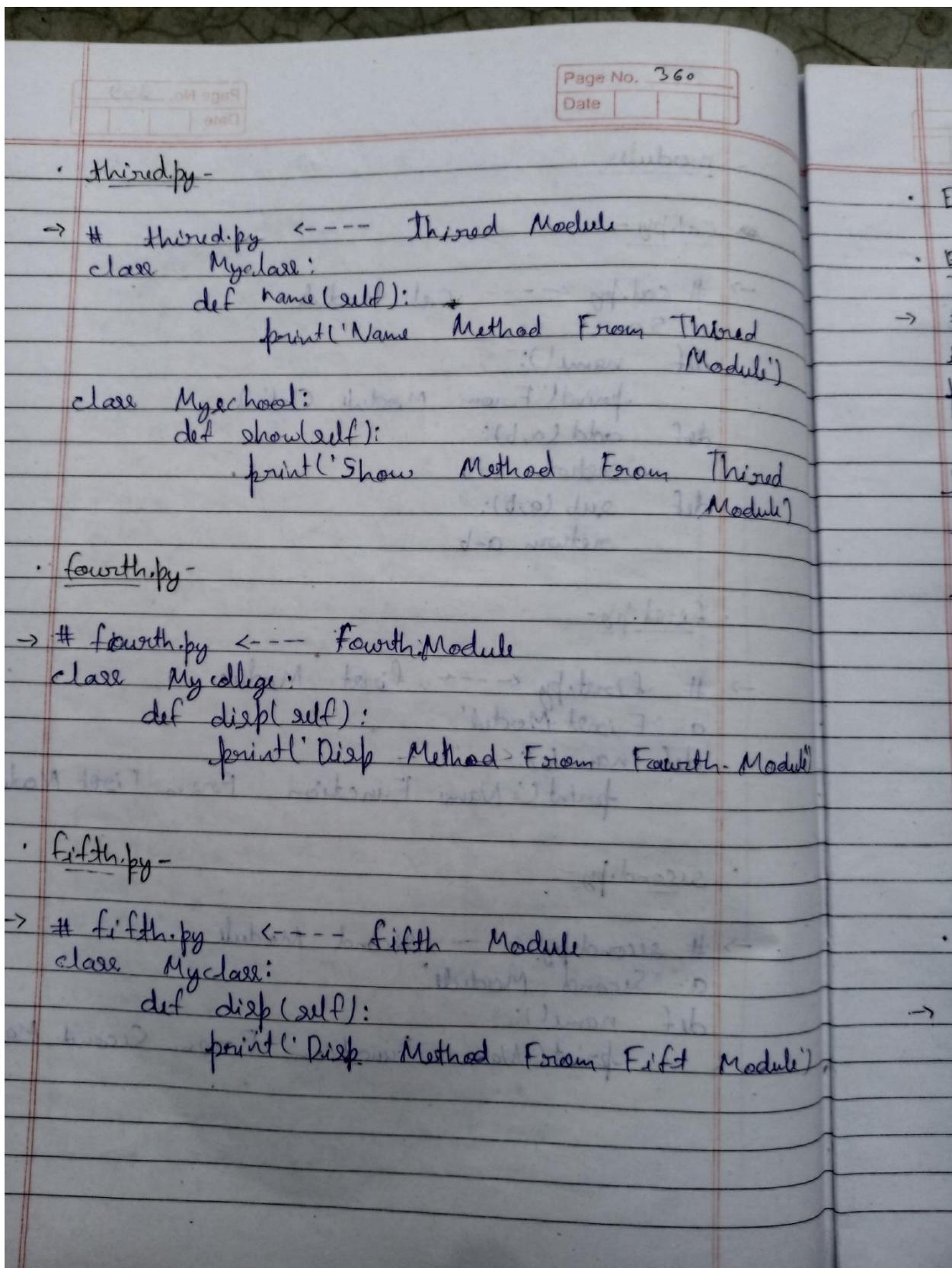
`example-9.py`

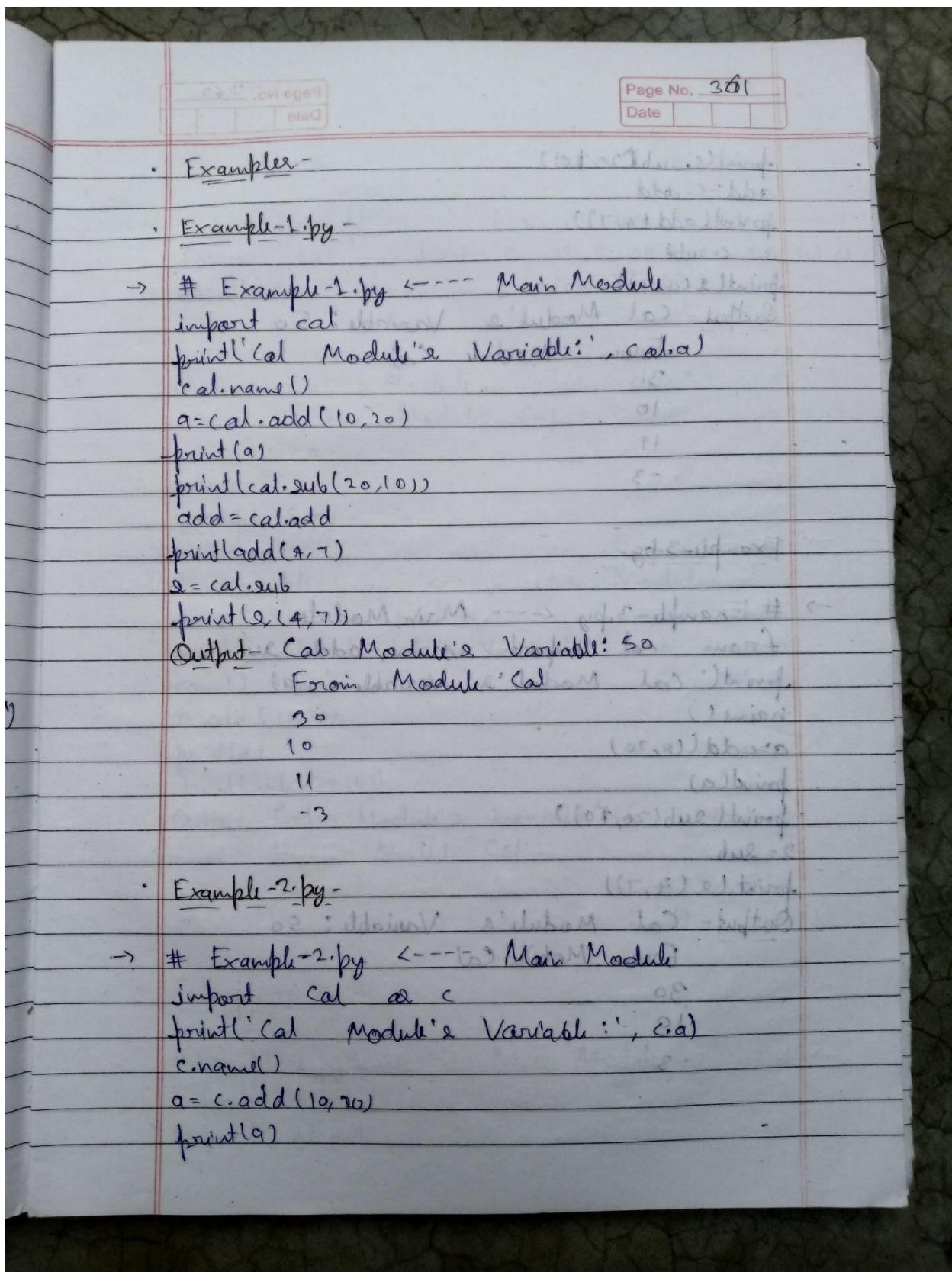
example - Lo.py

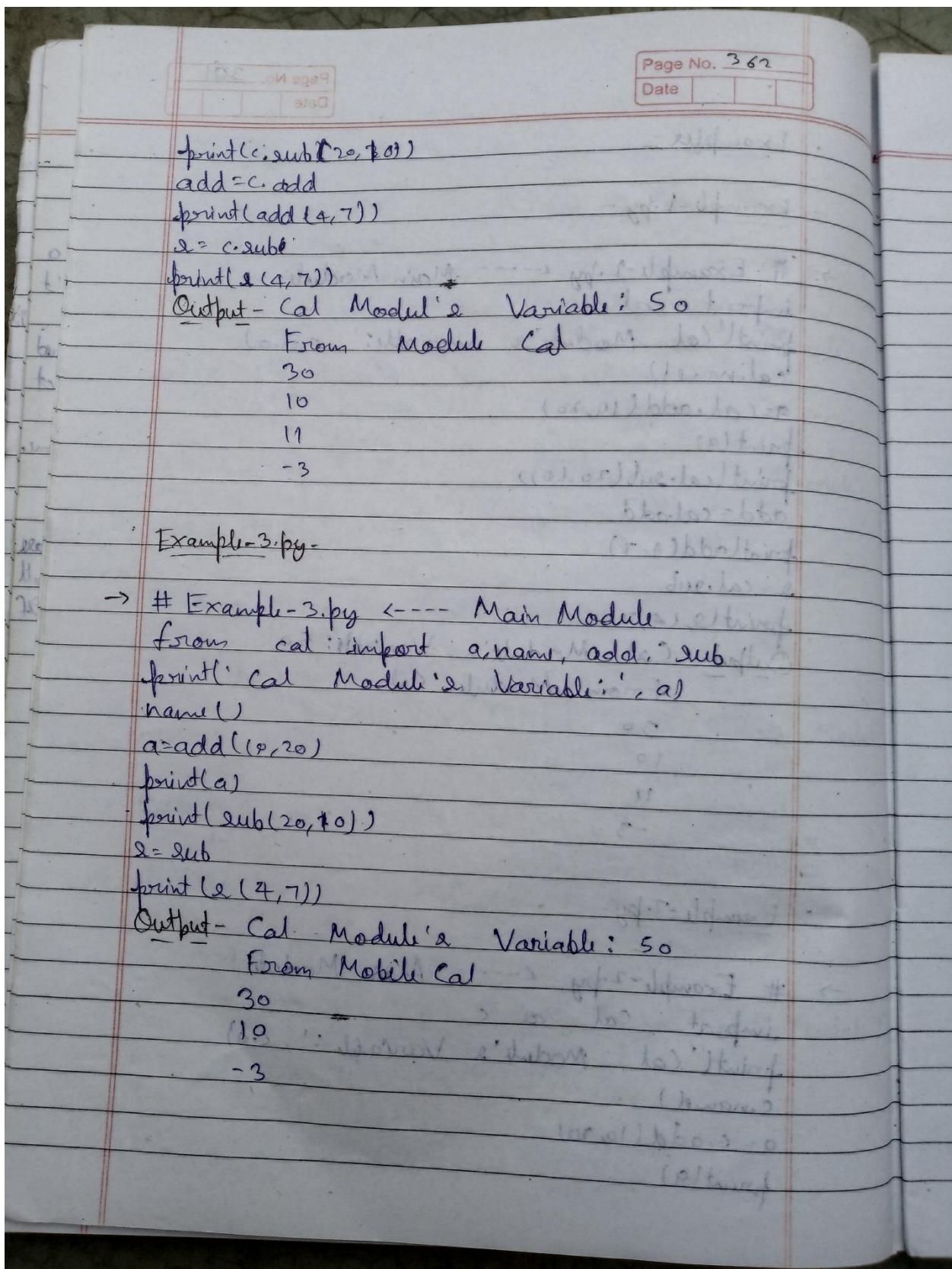
11.37

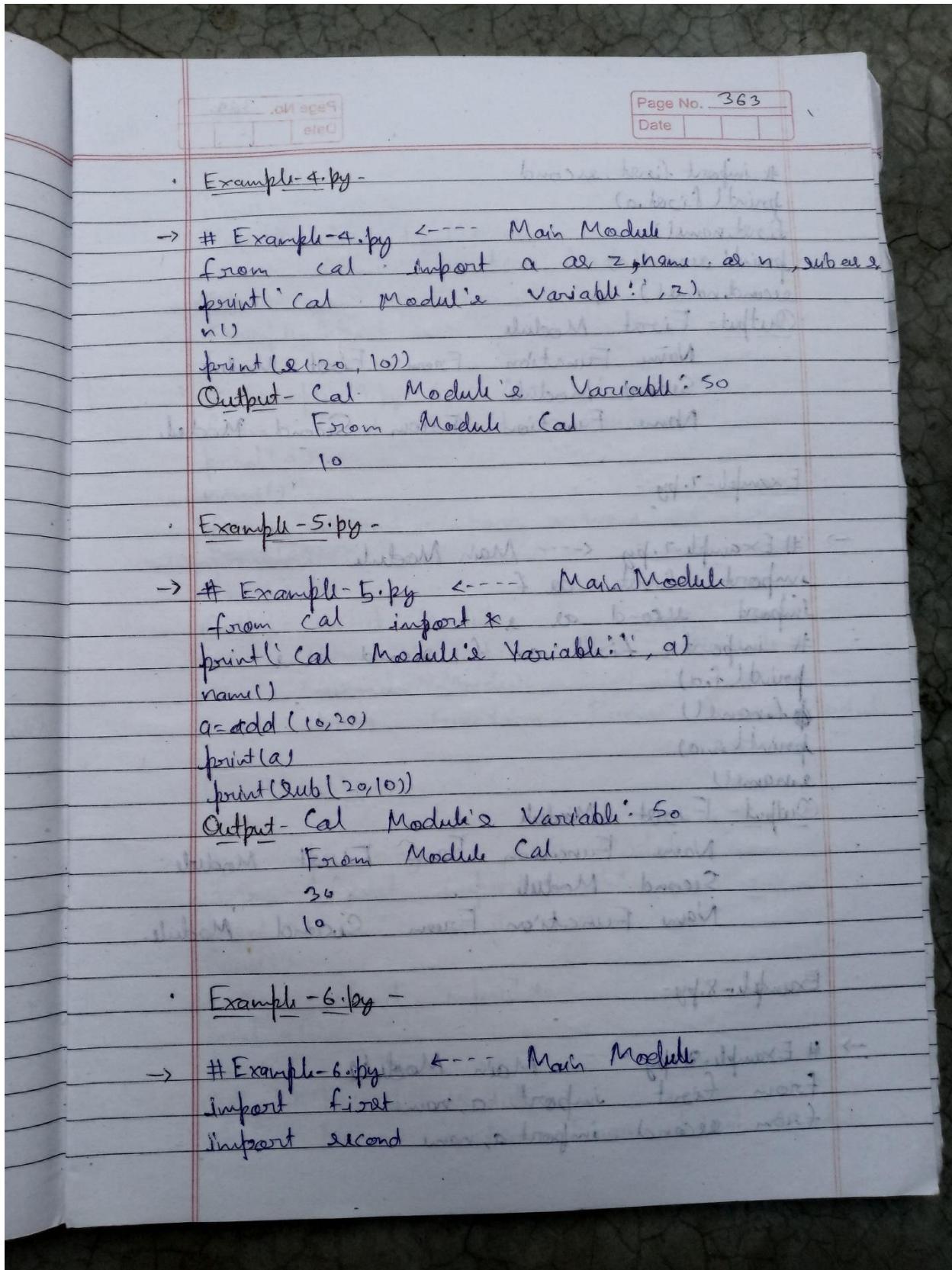
— 1 —

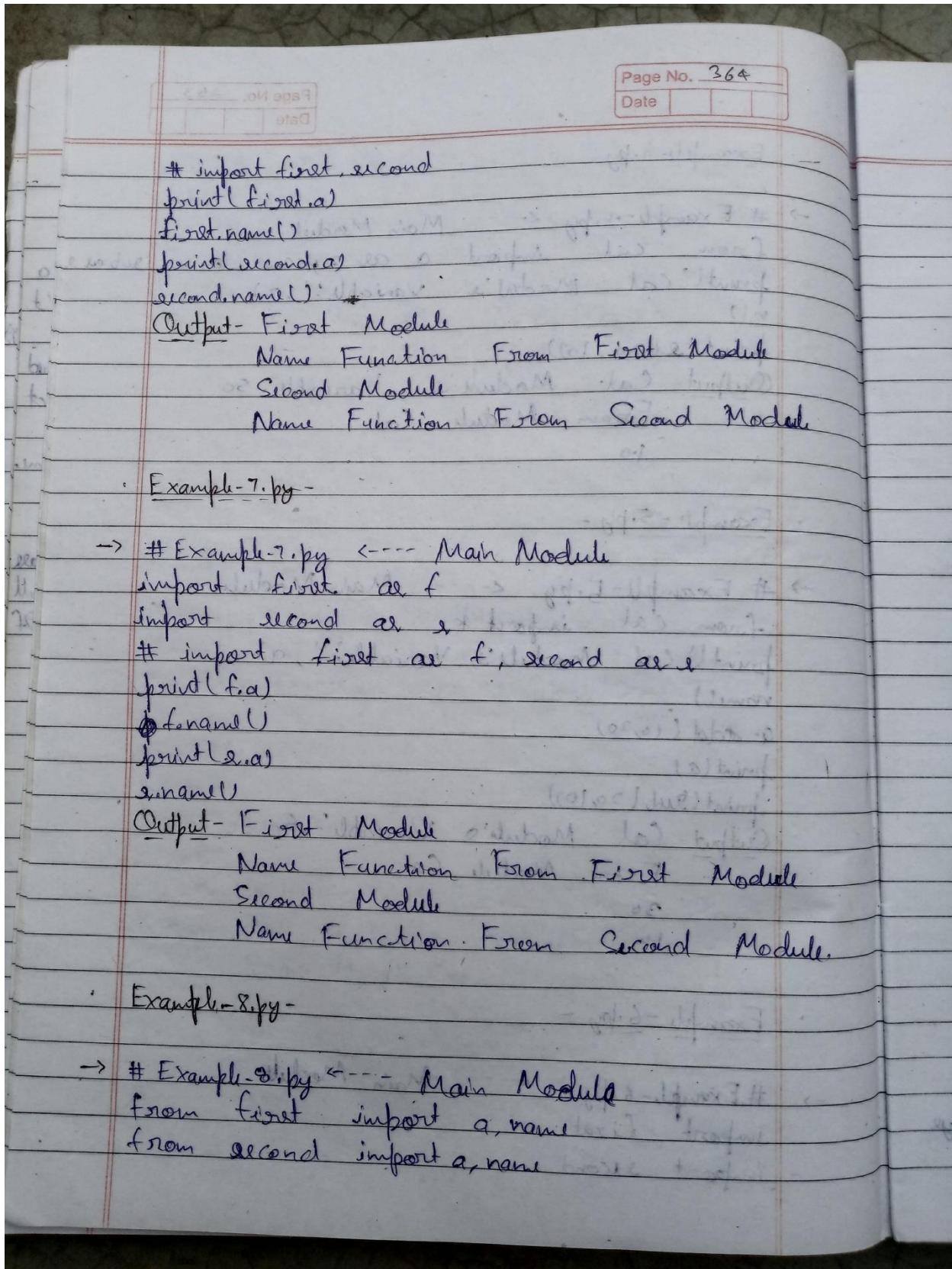


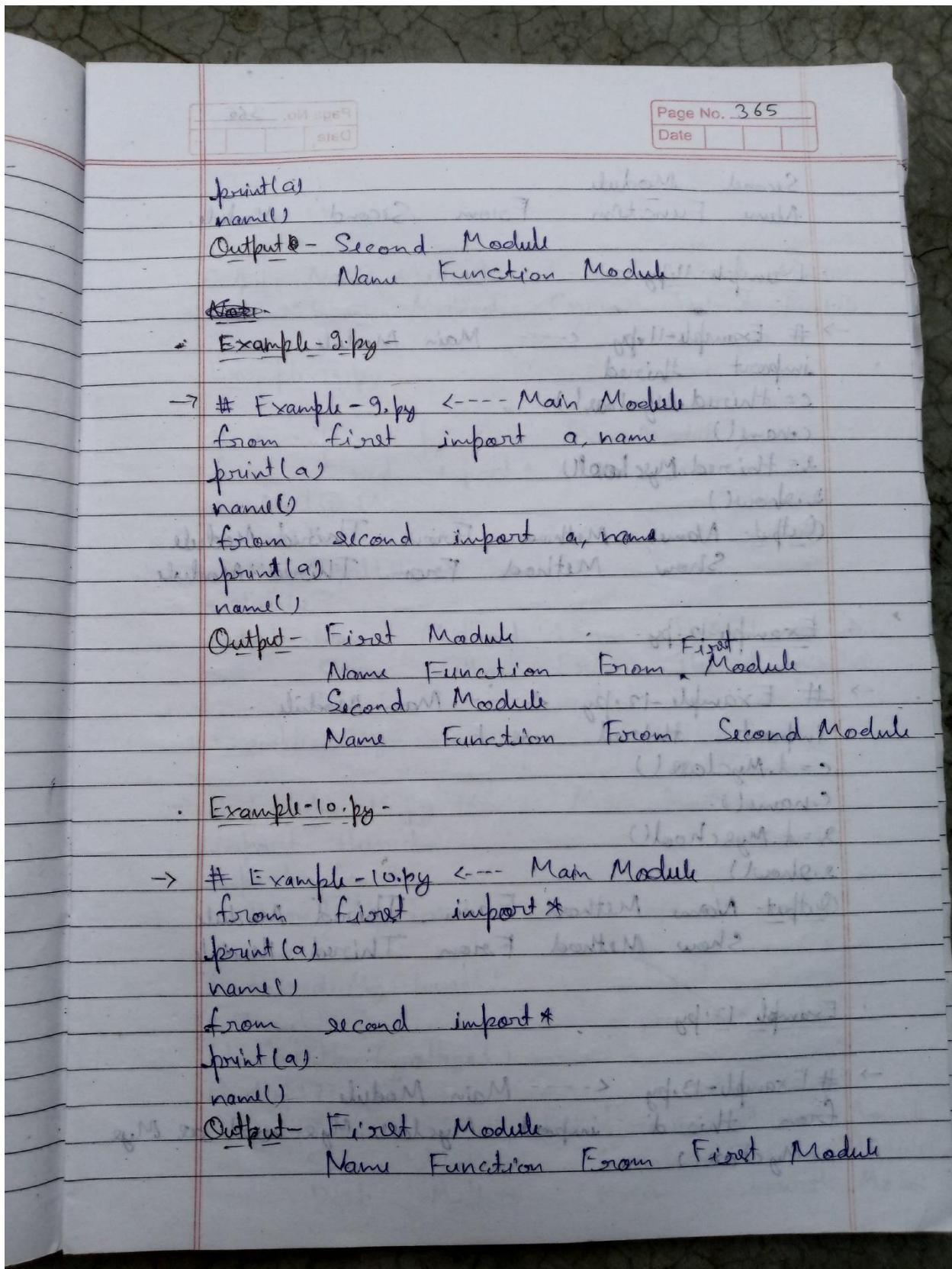


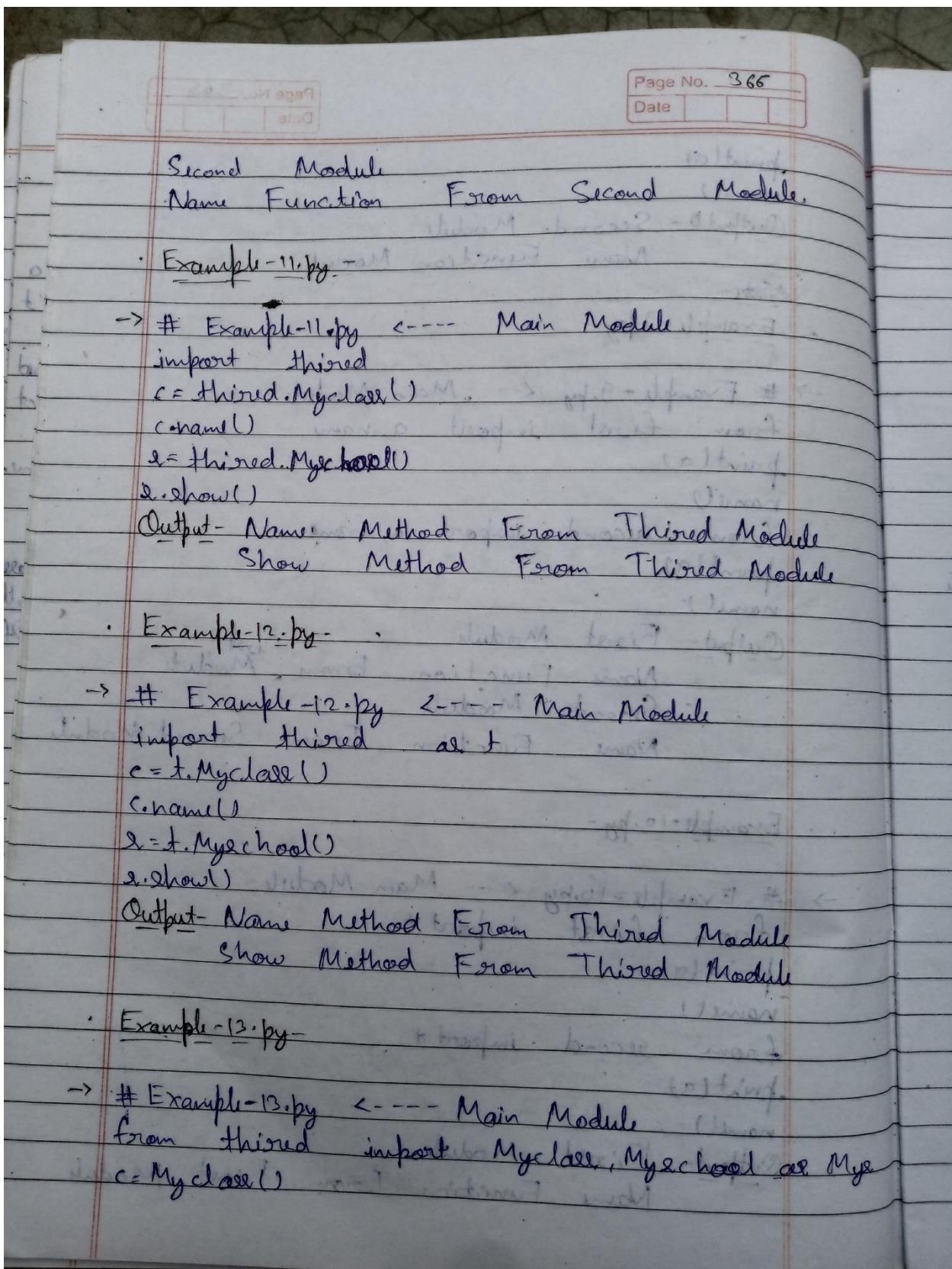


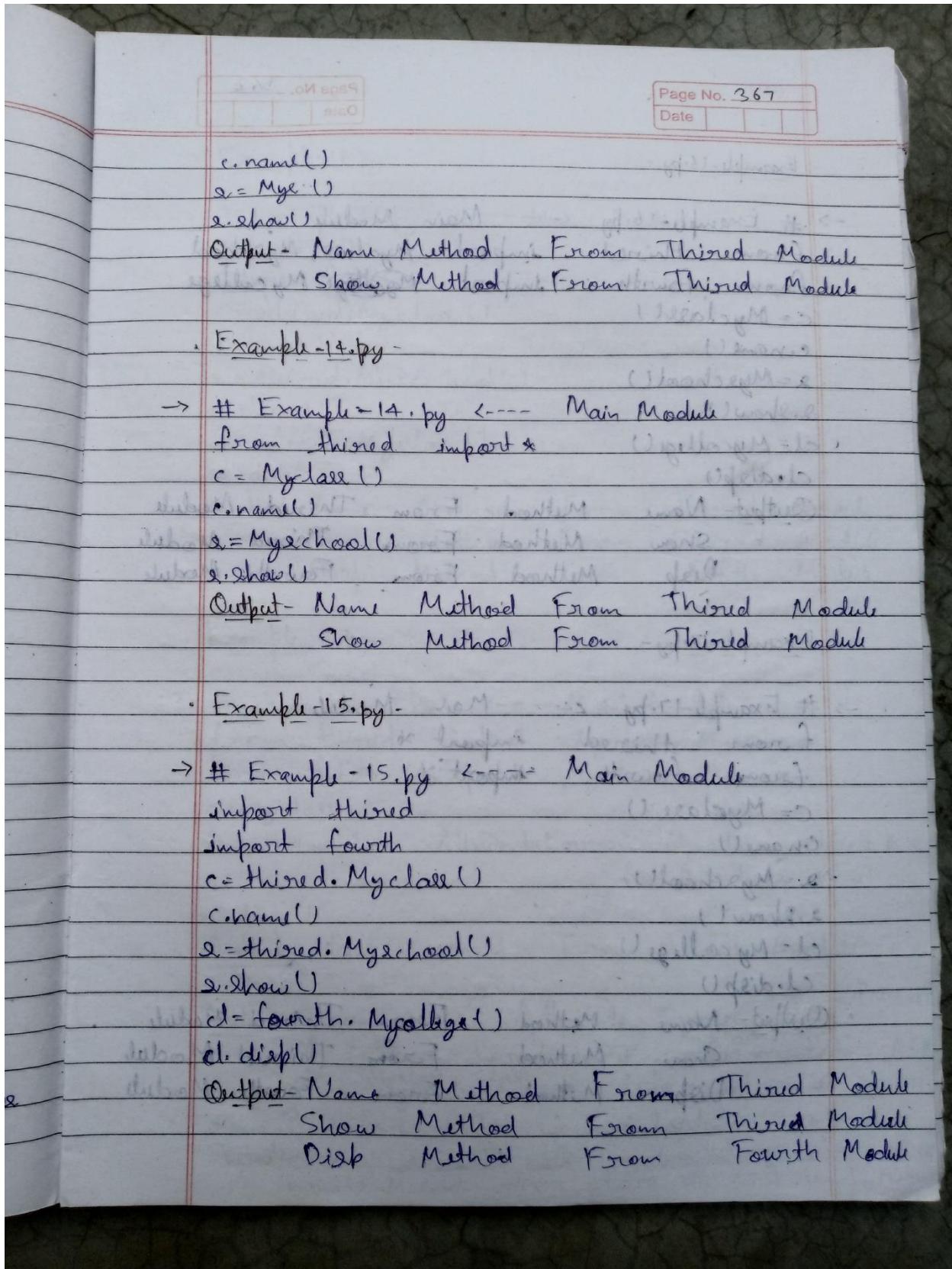












Page No. 368  
Date

Example-16.py -

```

→ # Example-16.py ---- Main Module
from third import Myclass, Myschool
from fourth import Mycollege
c = Myclass()
c.name()
s = Myschool()
s.show()
cl = Mycollege()
cl.disp()

```

Output- Name Method From Third Module  
 show Method From Third Module  
 Disp Method From Fourth Module

Example-17.py -

```

→ # Example-17.py ---- Main Module
from third import *
from fourth import *
c = Myclass()
c.name()
s = Myschool()
s.show()
cl = Mycollege()
cl.disp()

```

Output- Name Method From Third Module  
 show Method From Third Module  
 Disp Method From Fourth Module

Page No. 369  
Date

Example-18.py -

```

→ # Example-18.py  ←--- Main Module
import third
import fifth
c = third.Myclass()
c.name?
e = third.Myschool()
e.show()
cl = fifth.Myclass()
cl.disp()

```

Output - Name Method From Third Module  
 Show Method From Third Module  
 Disp Method From Fifth Module

Example-19.py -

```

→ # Example-19.py  ←--- Main Module
from third import Myclass, Myschool
from fifth import Myclass
c = Myclass()
# c.name() : AttributeError: 'Myclass' object has no attribute 'name'
c.disp()

```

Output - Disp Method From Fifth Module

Note - c.name() : Attribute Error

It will throw Attribute Error because there is no name attribute in Myclass class.

class Myclass:
 def disp(self):
 print("Disp Method")

class Myschool:
 def show(self):
 print("Show Method")

class Myclass:
 def disp(self):
 print("Disp Method")

class Myschool:
 def show(self):
 print("Show Method")

Page No. 370  
Date

Example-20.py -

```

→ # Example-20.py      --- Main Module
from third import MyClass, MyClass
c = MyClass()
c.name()
from fifth import MyClass
cl = MyClass()
cl.disp()

```

Output - Name Method From Third Module  
 Disp Method From Fifth Module.

- Module Search Path

Ex When a module named cal imported,  
 the interpreter first searches for a  
 built-in module with that name.  
 If not found, it then searches for  
 a file named cal.py in a list of  
 directories given by the variable  
`sys.path`.

Note `sys.path` is initialized from these  
 locations -

- Current Directory
- If not found then searches each directory  
 in the list variable `PYTHONPATH`
- If not found then searches installation  
 dependent default path.
- `PYTHONPATH` is a list of directory names, with  
 the same syntax as the list variable `PATH`.  
(Note: Python install \$ is in lib folder)

Package

Page No.	371
Date	

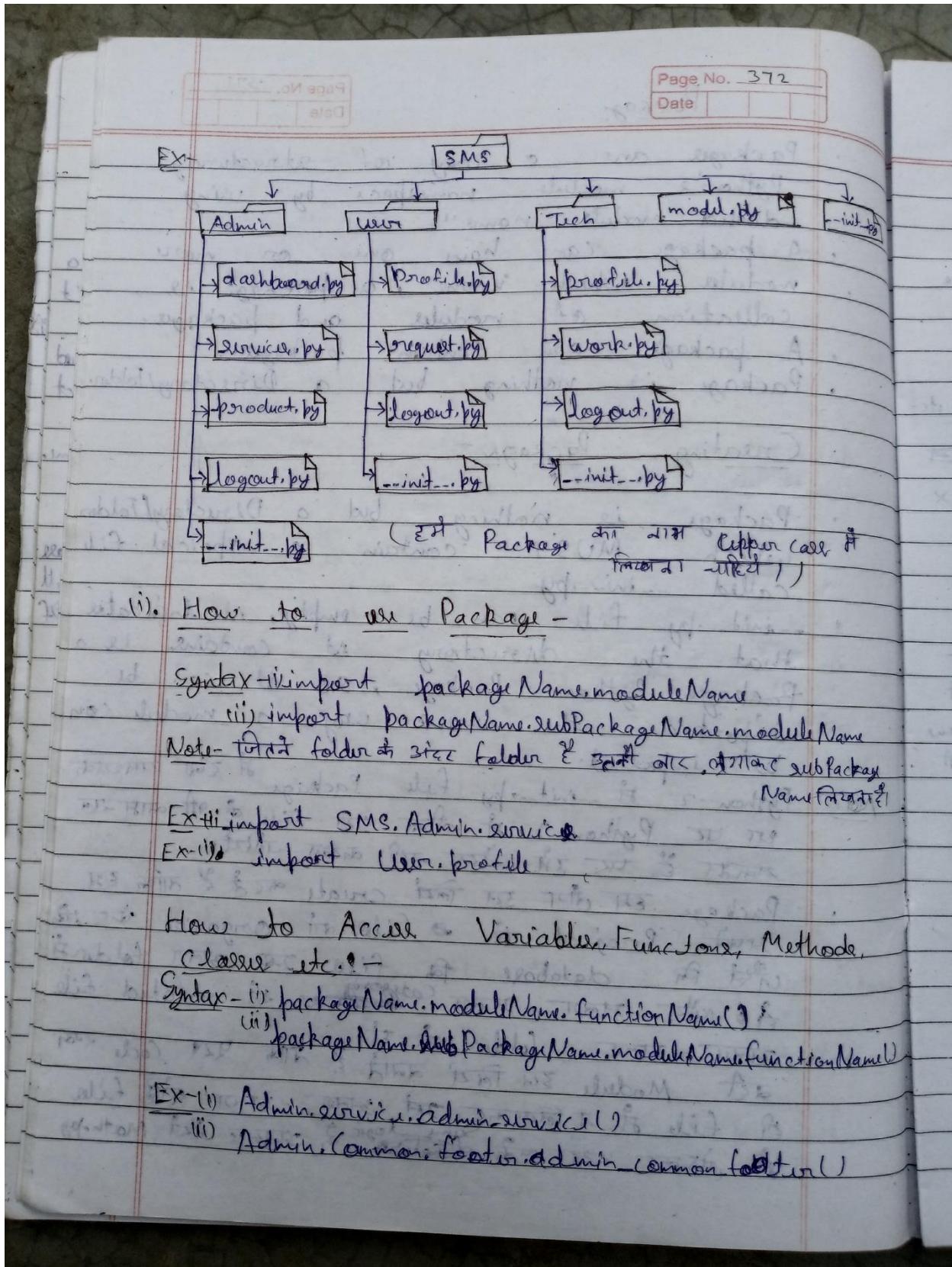
- Packages are a way of structuring Python's module namespace by using "dotted module names".
- A package can have one or more modules which means, a package is a collection of modules and packages.
- A package can contain packages.
- Package is nothing but a Directory/Folder.

Creating Package -

- Package is nothing - but a Directory/Folder which MUST contain a special file called `__init__.py`
- `__init__.py` file can be empty, it indicates that the directory it contains is a Python Package, so it can be imported the same way a module can be imported.

Note- Python 2 में `__init__.py` file Package में होना आवश्यक नहीं है Python 3 में जीवा इस file के शी काम नहीं करता है पर ही यहाँ ऐसा बड़ी कठोर गारियाँ।

• Package हम तोग इस लिए create करते हैं ताकि हम Project में file को organize कर सकें कि database file वह अलग folder में ही रहे और अलग - अलग Category के related file अलग - अलग folder में ही। वह Module जैसे ही करते हैं तो कि परा Code उसी file में बदलना करते हैं अलग - अलग एक file में तो कर दें जैसे कि ~~some logic~~ के अलावा भी `Math.py`



Page No. 373  
Date

(iii) How to use Package -

Syntax- (i) from packageName import moduleName  
(ii) from packageName.subPackageName import moduleName

Ex-(i) from Admin import service  
(ii) from Admin.Common import footer.

How to Access Variable, Function, Method, Class, etc. -

Syntax- module Name.function Name()

Ex- service.admin.service()

(iii) How to use Package -

Syntax- (i) from packageName.moduleName import funName  
(ii) from packageName.subPackageName.moduleName  
import funName

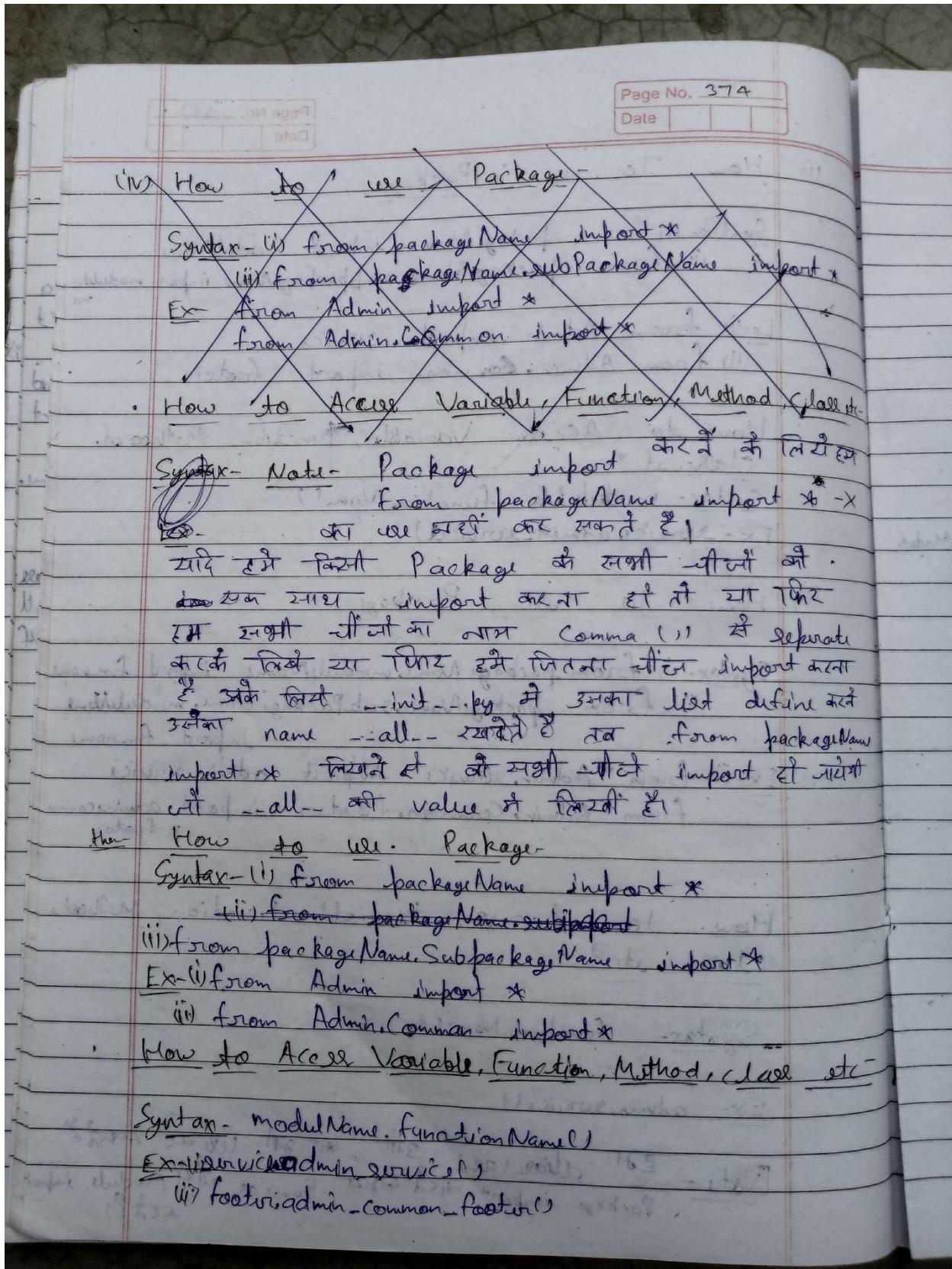
Ex-(i) from Admin.service import admin.service  
from Admin.Common.footer import admin.common.footer

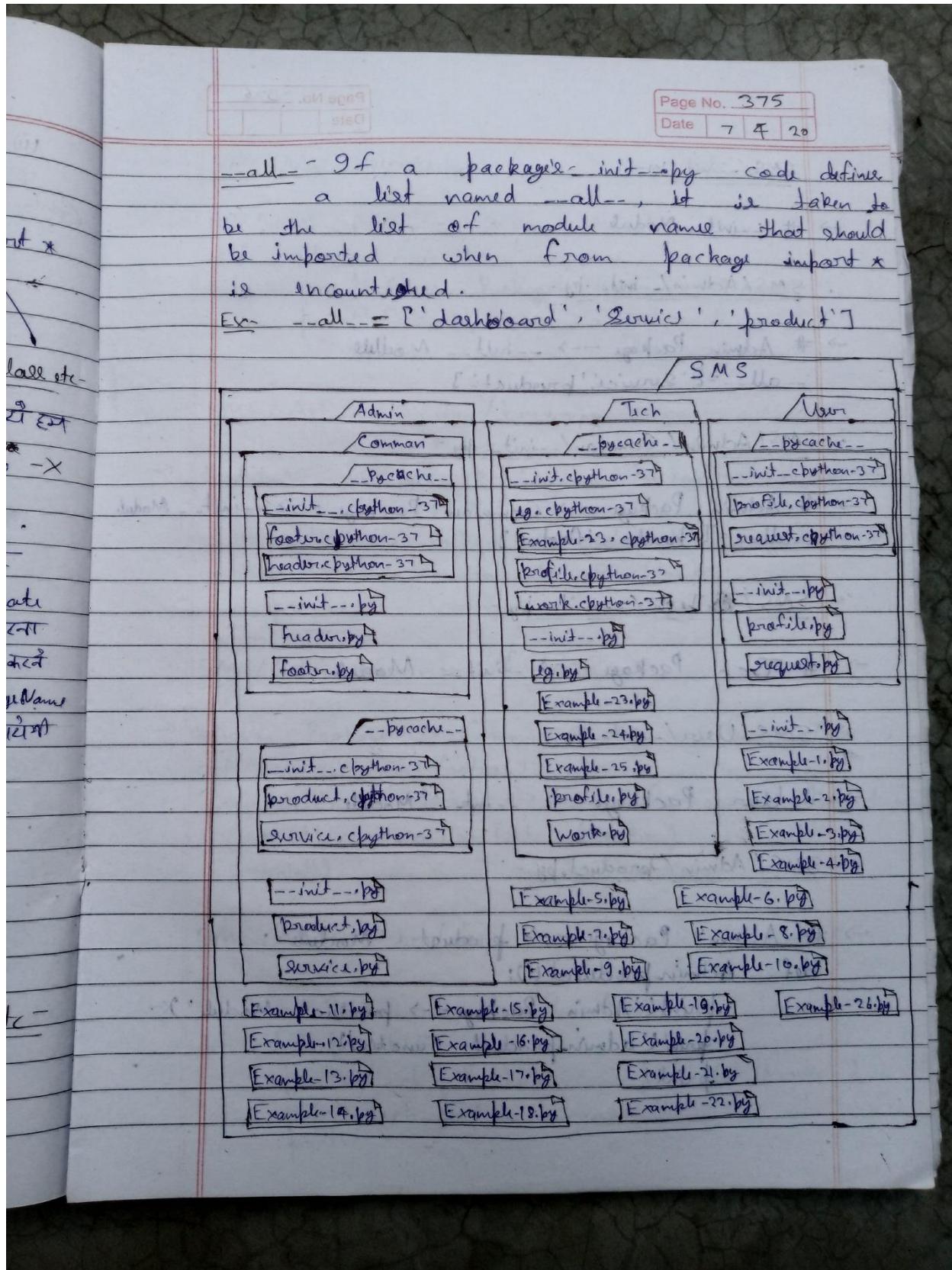
How to Access Variable, Function, Method, Class, etc. -

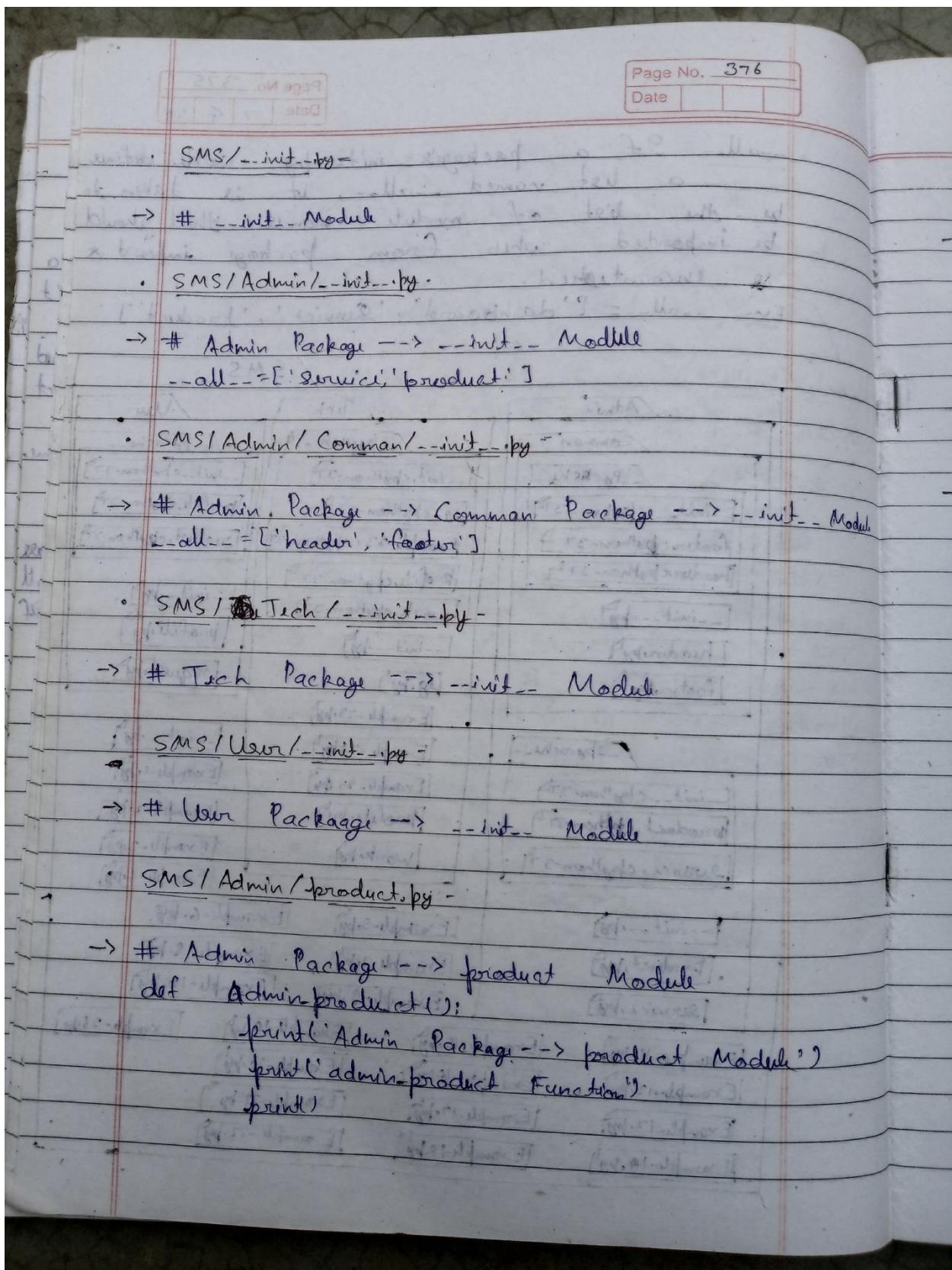
Syntax- function Name()

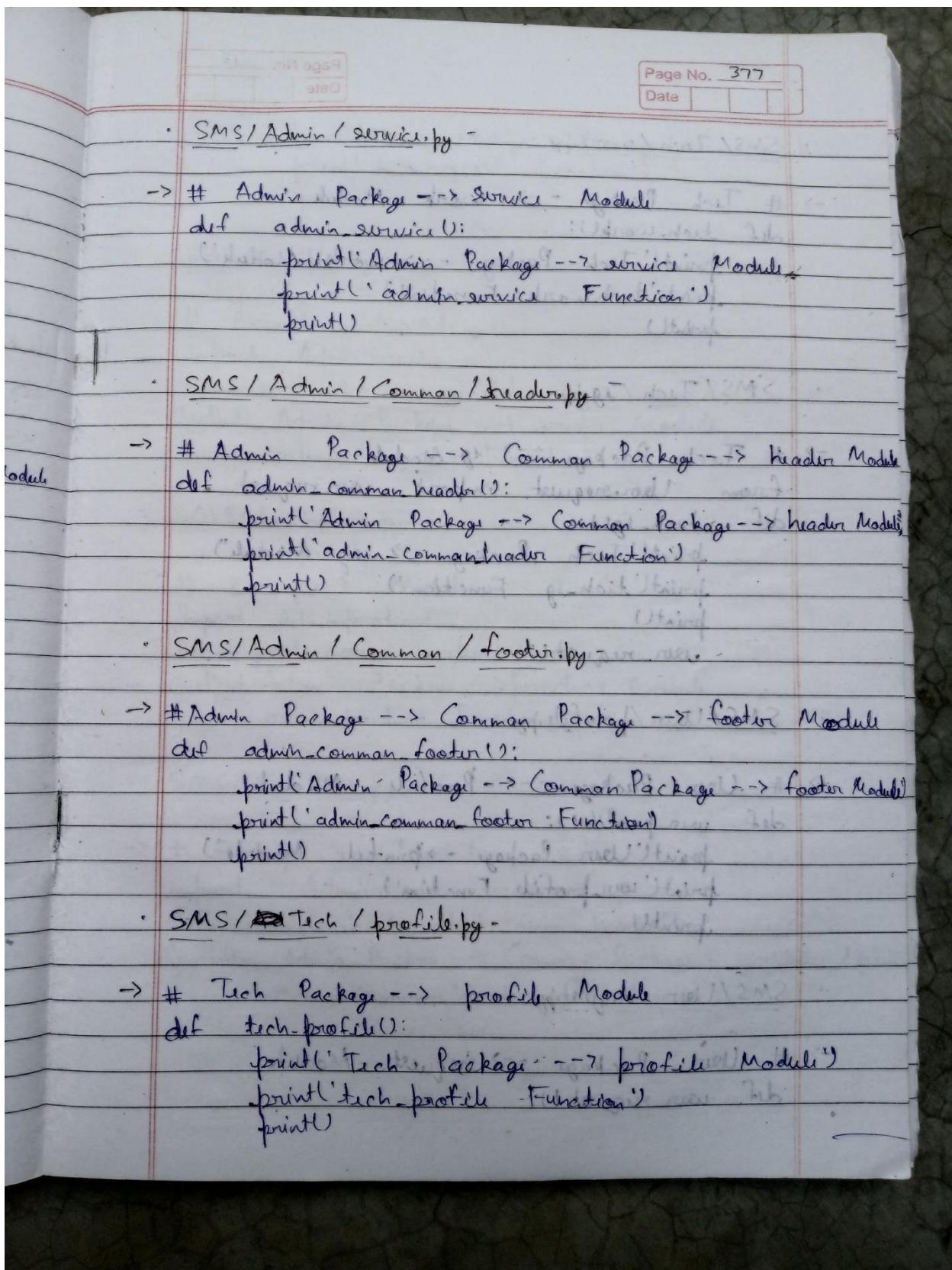
Ex- admin.service()

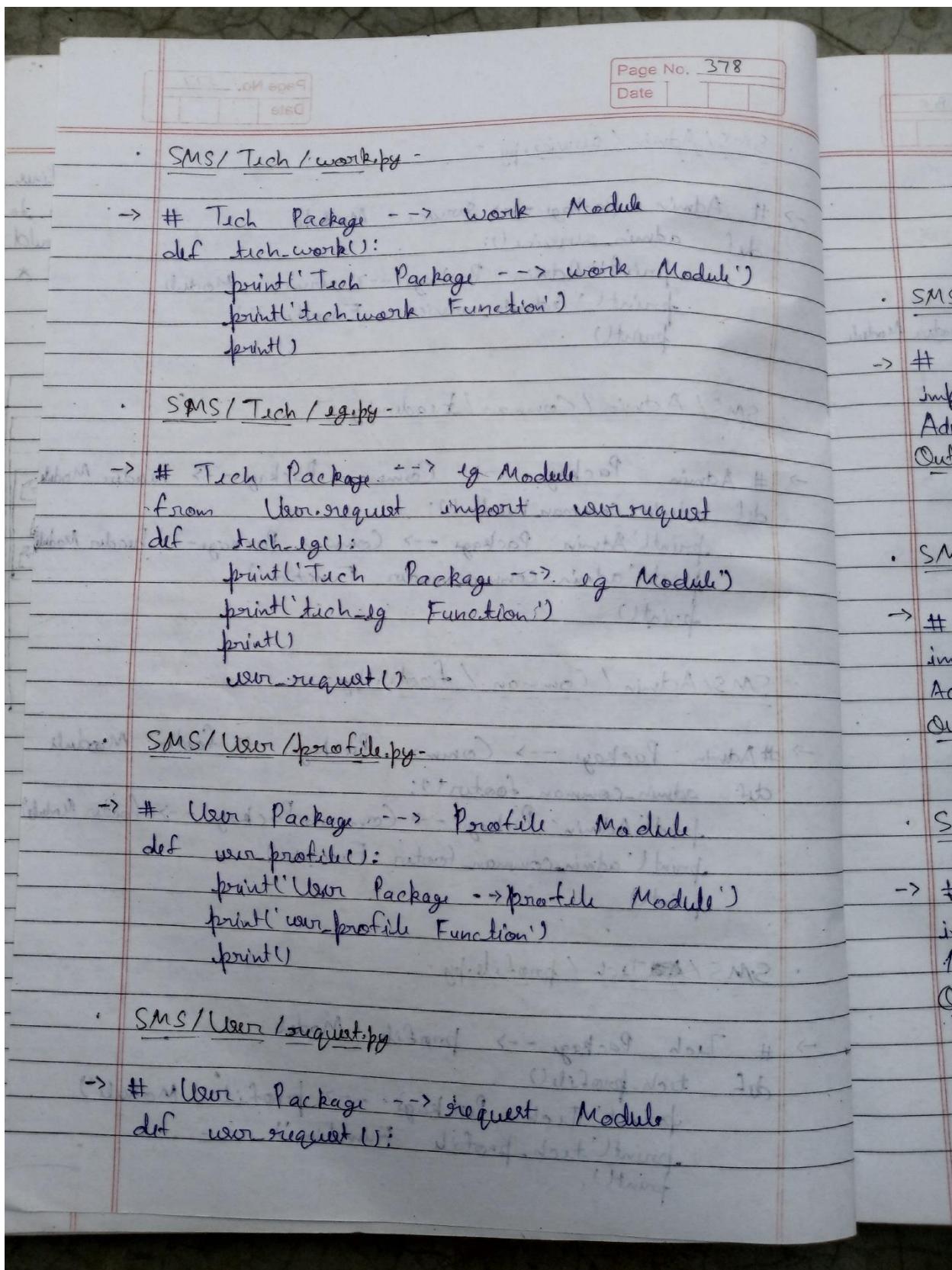
Note - It's alias (as), like we do import package import class or import same module import class

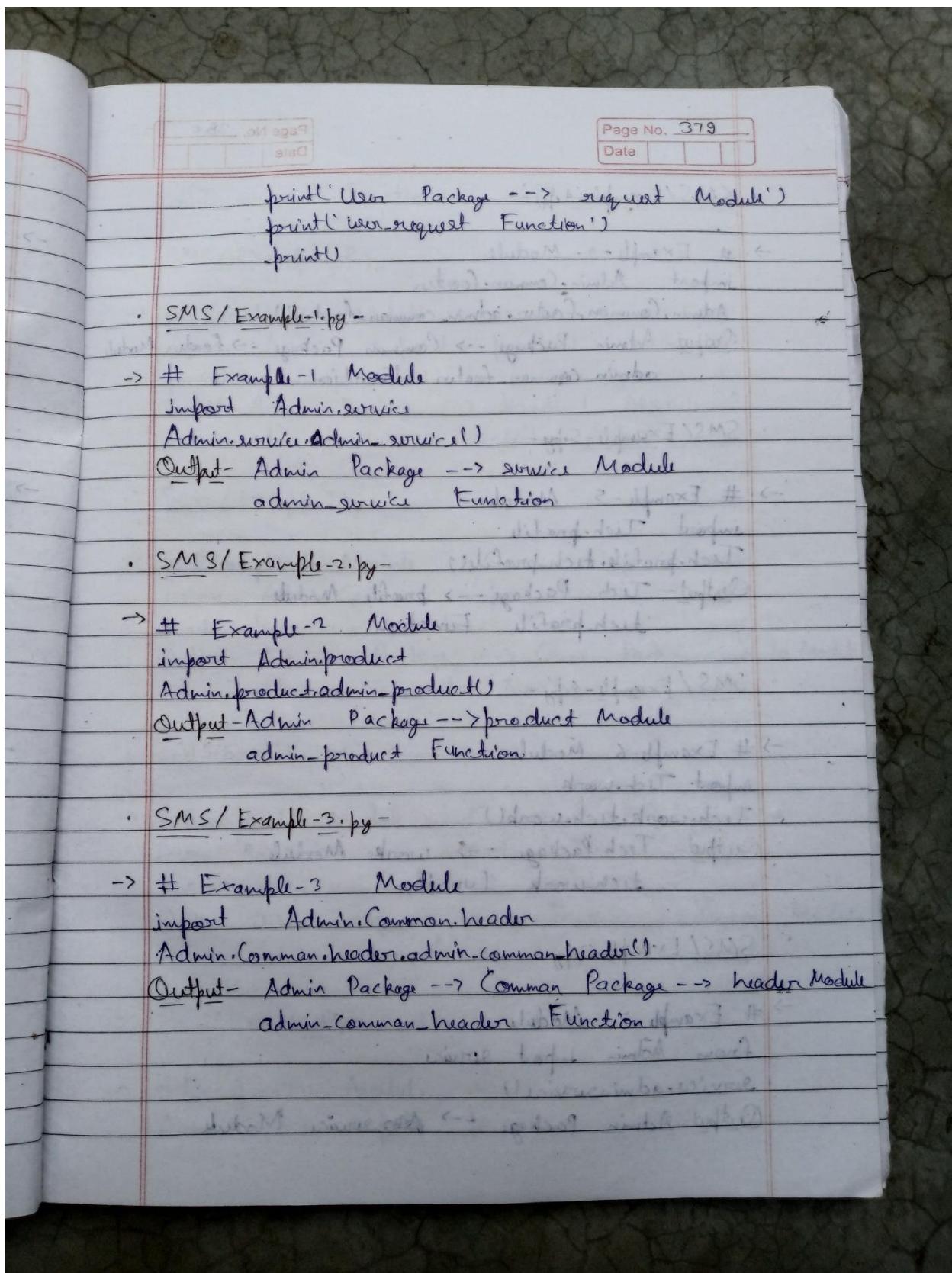


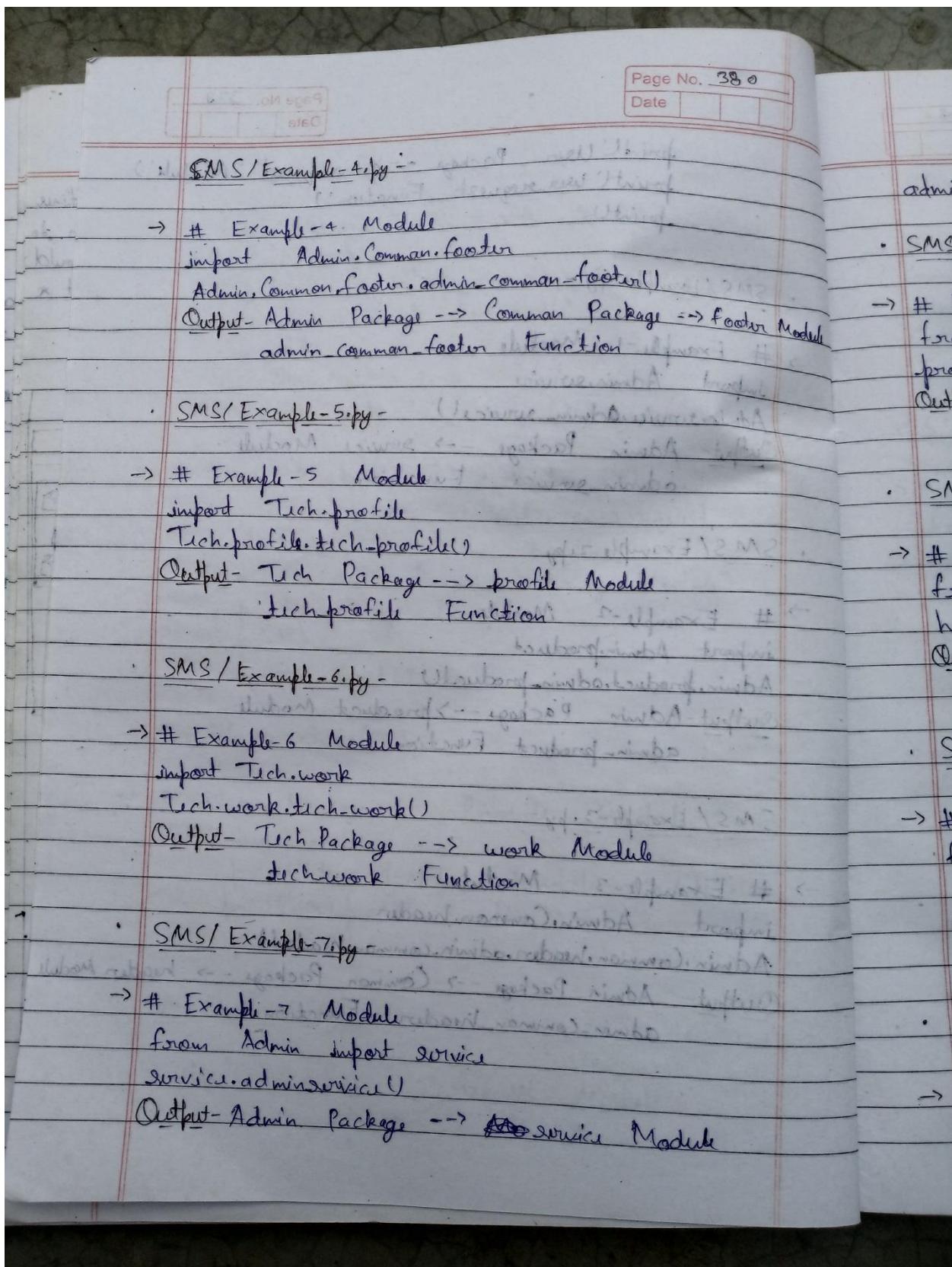


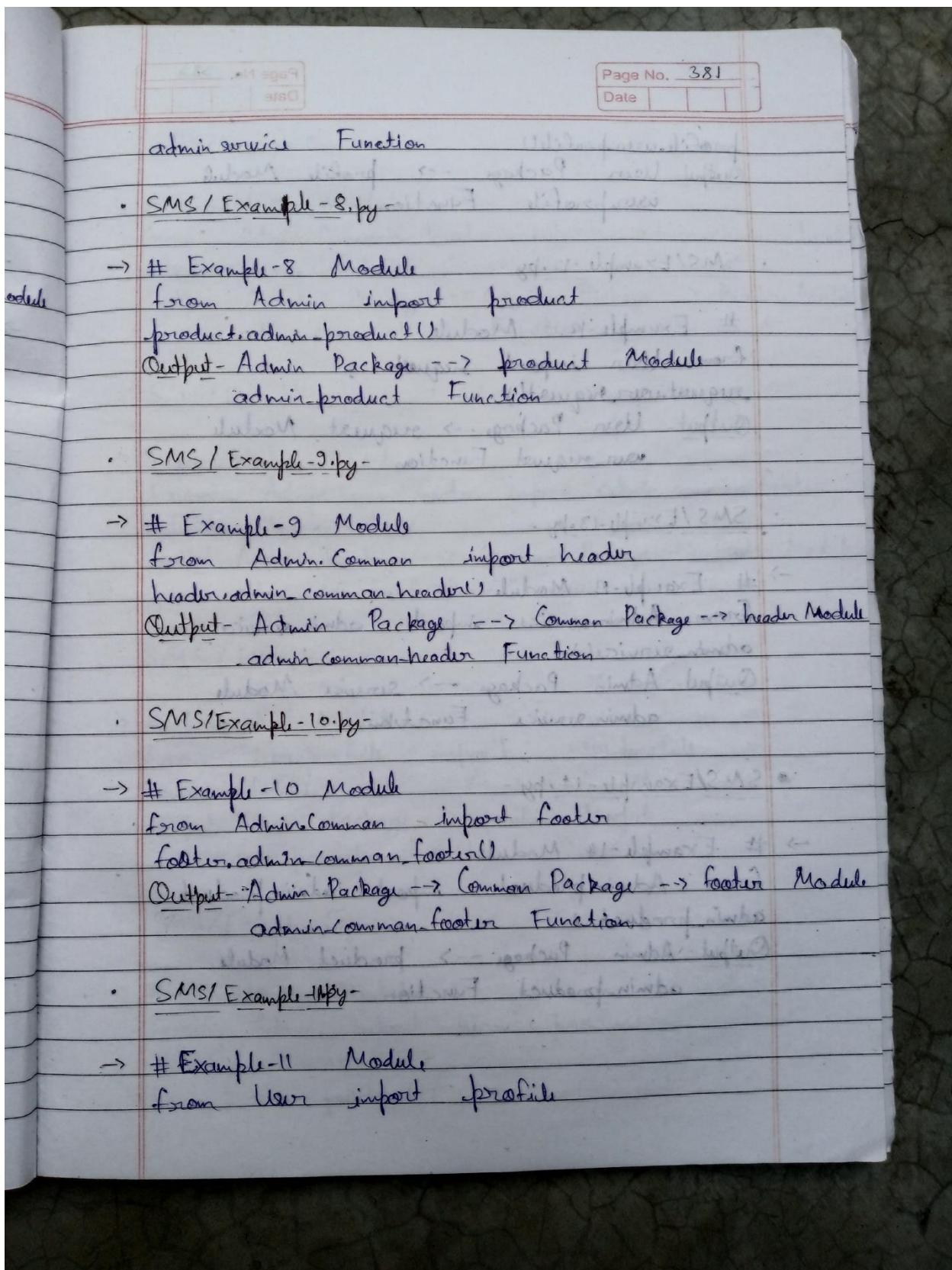


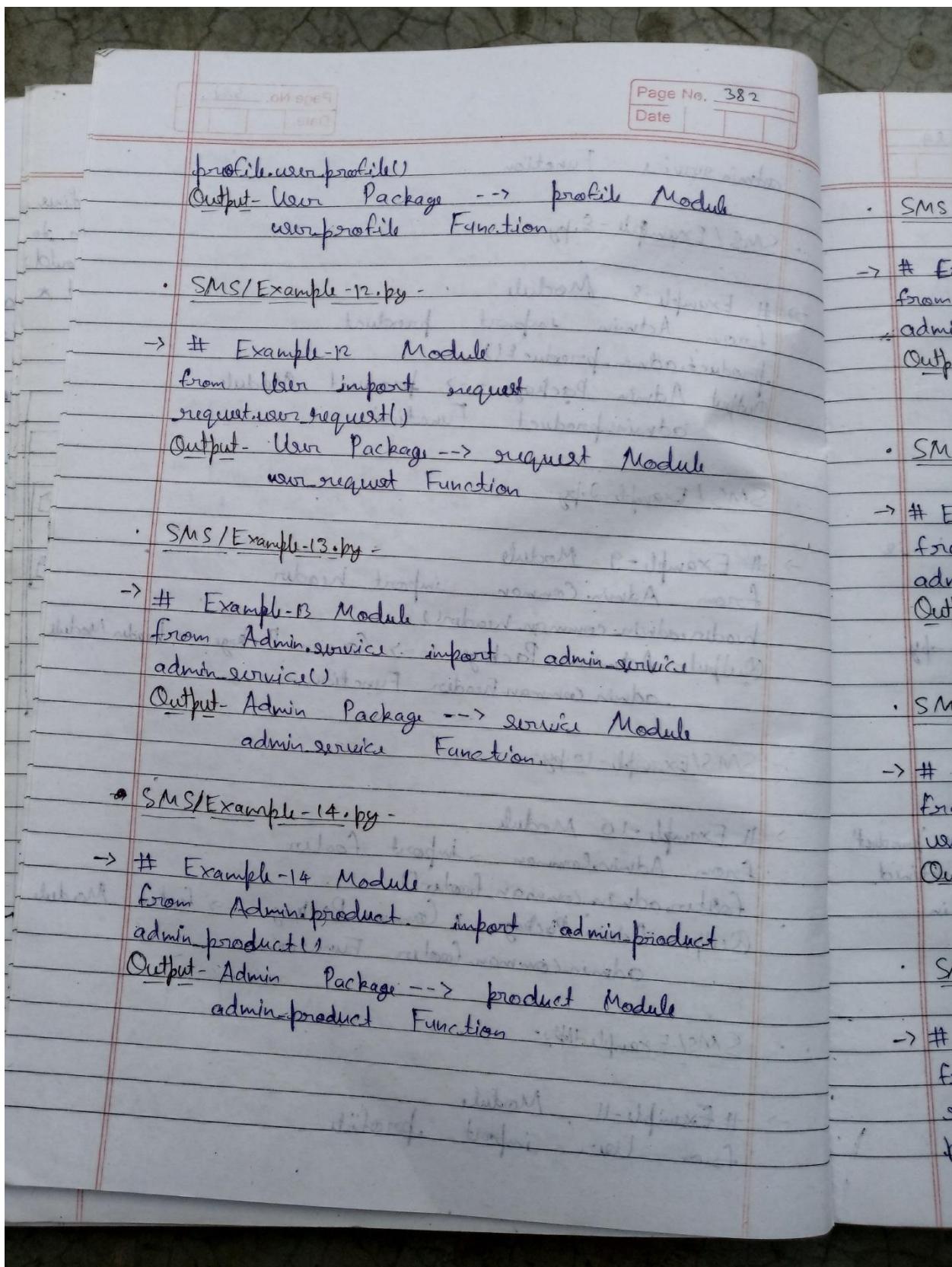


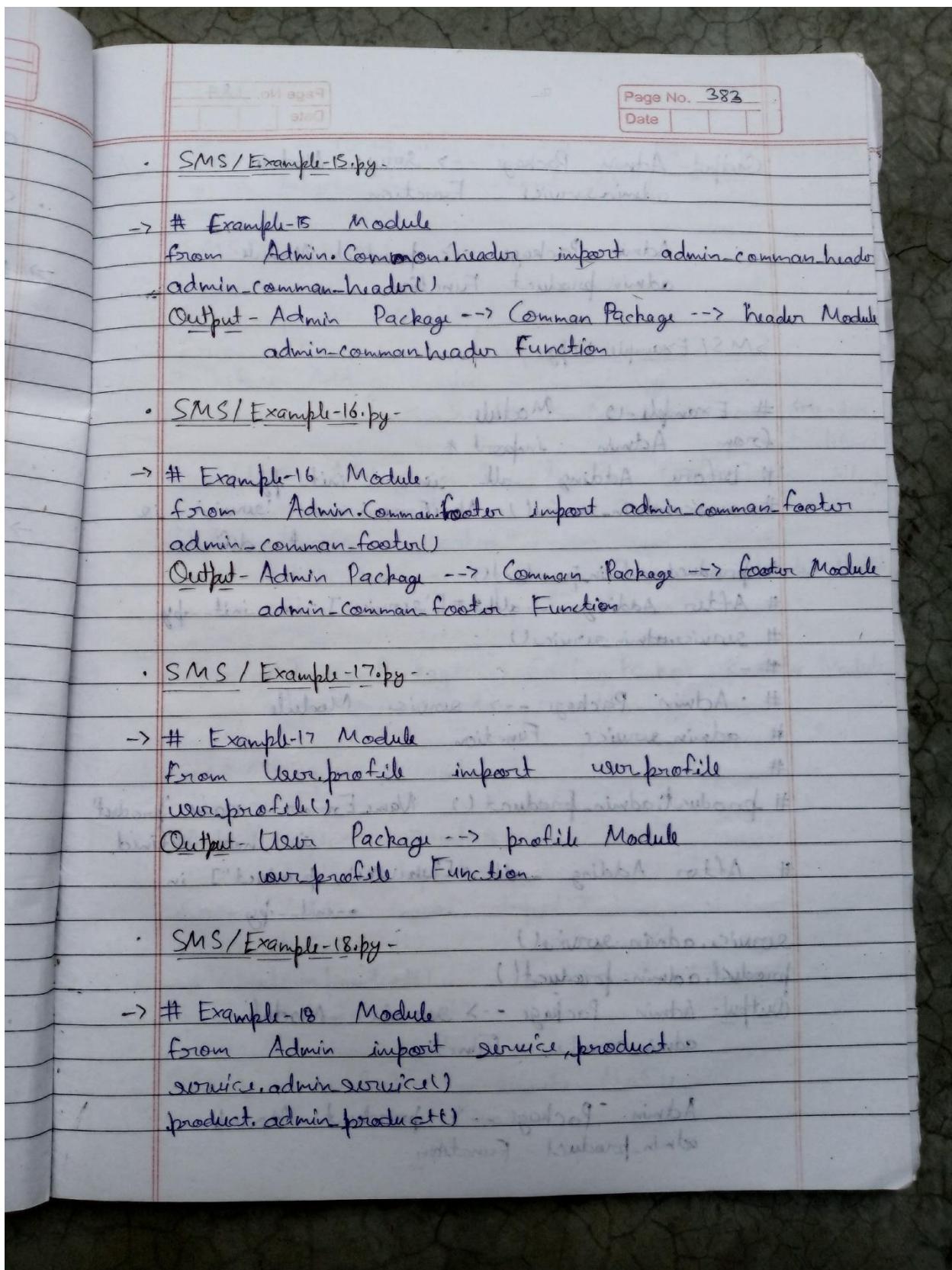


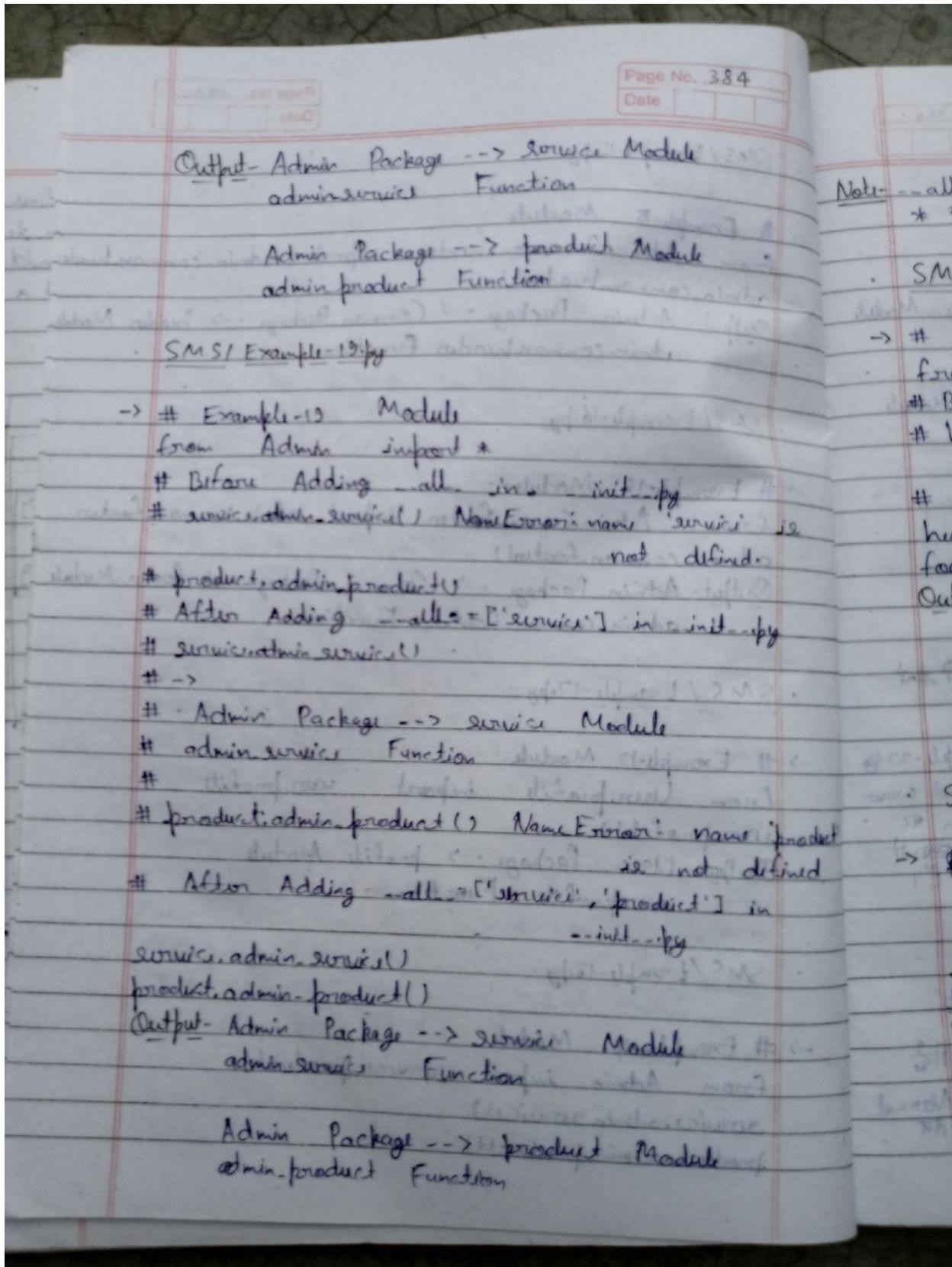


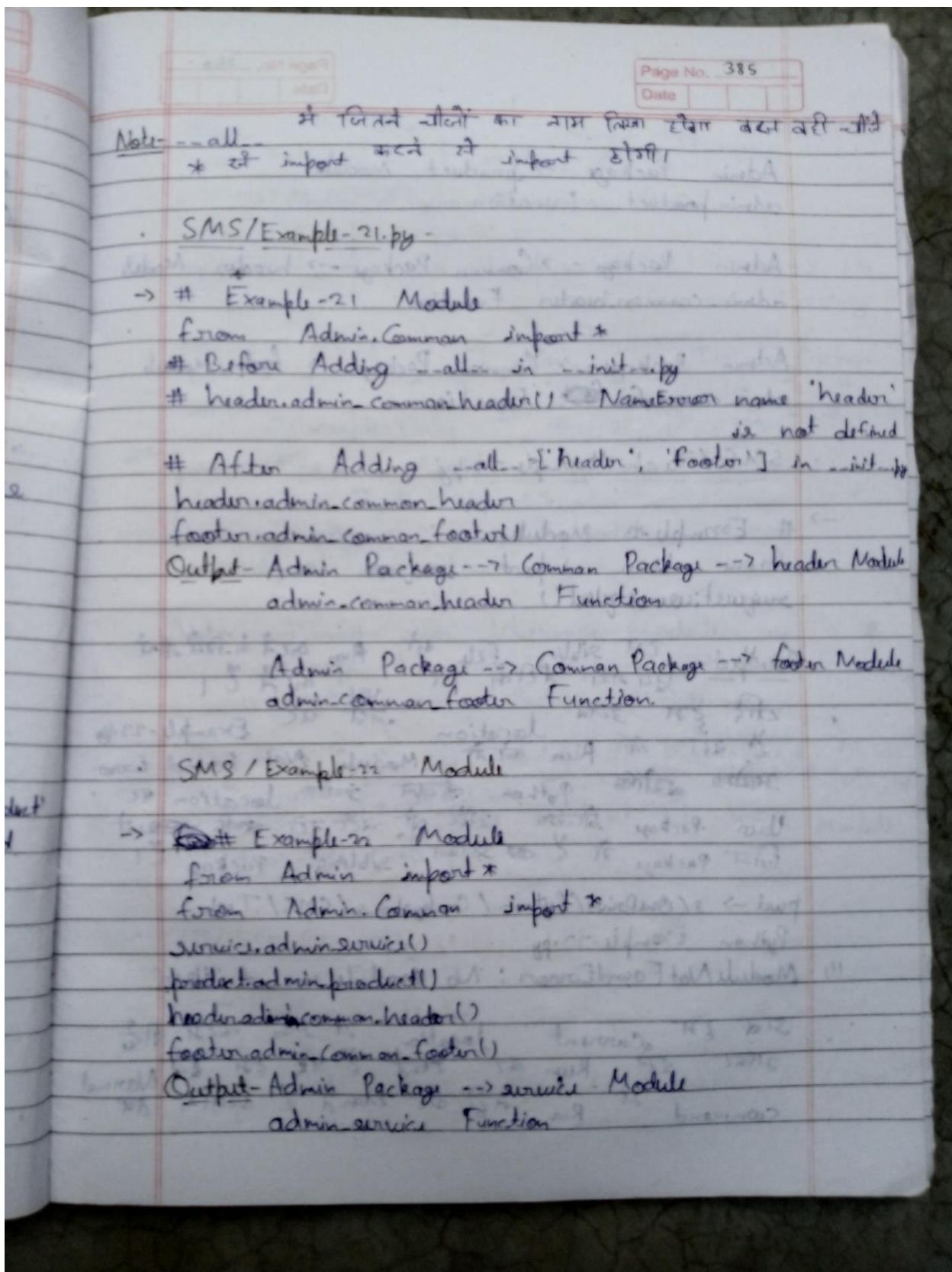


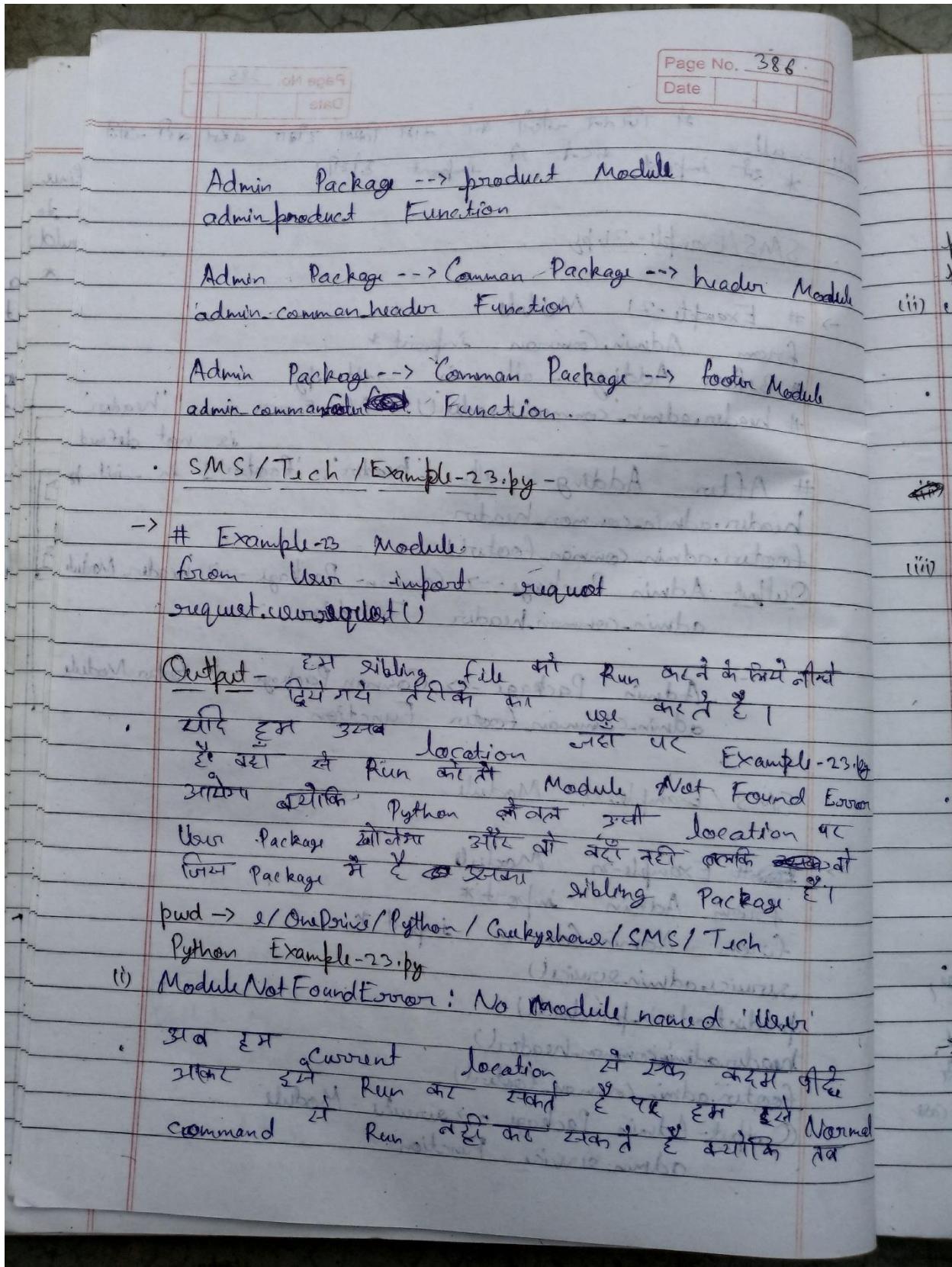


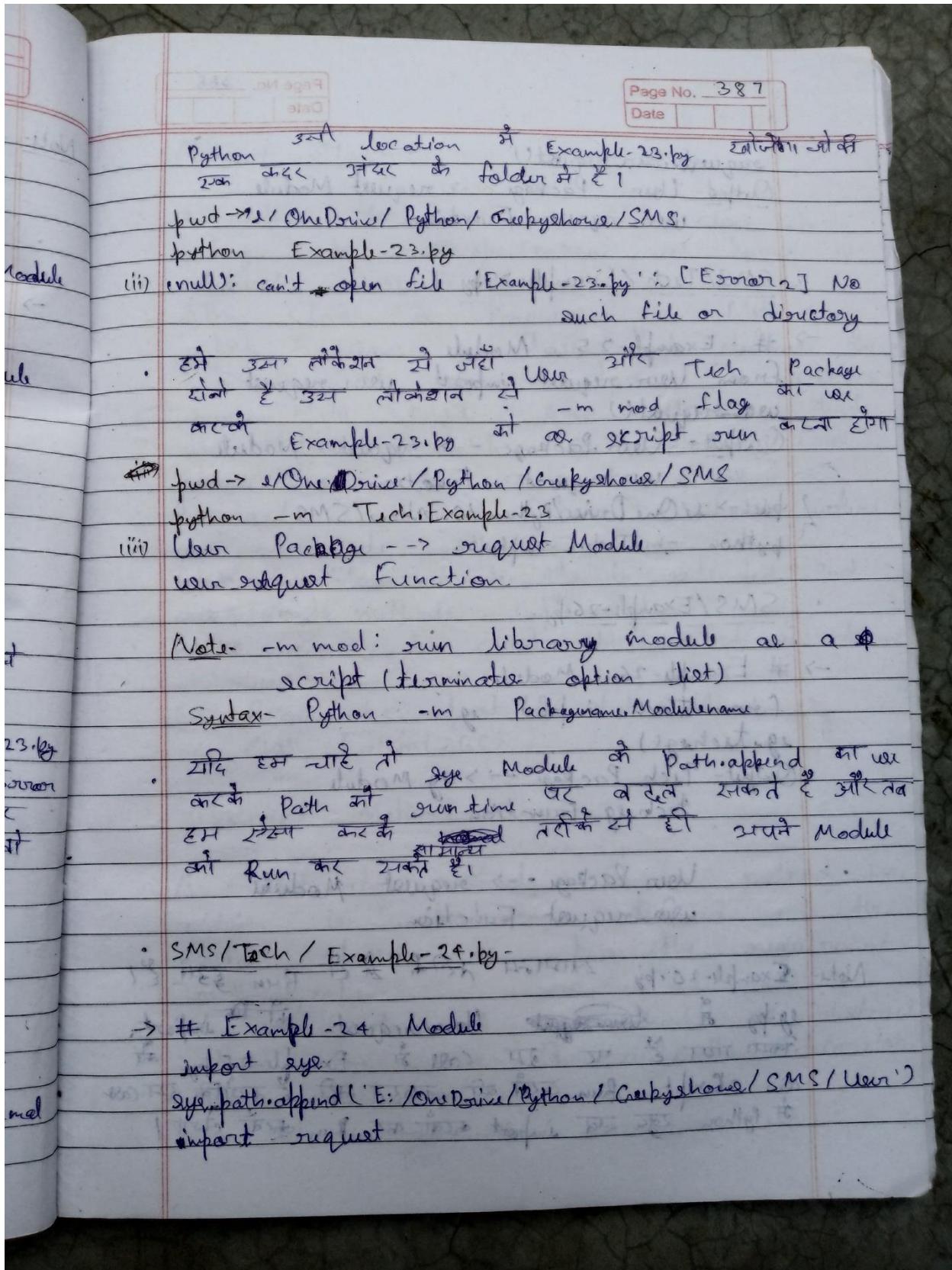












Page No. 388  
Date

`request.user.request()`

Output - User Package  $\rightarrow$  request Module  
user request Function

• SMS/Tech/Example-25.py

$\rightarrow$  # Example-25 Module  
from User.request import user\_request  
user\_request()

Output - User Package  $\rightarrow$  request Module  
user request Function

{  
} pud  $\rightarrow$  1/One Drive/Python/CrashCourse/MS  
python -m Tech.Example-25

• SMS/Example-26.py

$\rightarrow$  # Example-26 Module  
from Tech import eg  
eg.techeg()

Output - Tech Package  $\rightarrow$  eg Module  
tech-eg Function

User Package  $\rightarrow$  request Module  
user request Function.

Note - Example-26.py परामर्श नहीं किया गया Run होता है।  
eg.py में ~~user.request~~ User.request को import किया गया है पर इस call में Example-25.py को as script run करने की प्रक्रिया है जोकि इस call में Python द्वारा सब import करो कर Run करते होगा।

Page No. 389  
Date 8 4 20

Abstract Class -

- A class derived from ABC class which belongs to ~~abc~~ abc module, is known as abstract class in Python.
- ABC class is known as Meta class which means a class that defines the behaviour of other classes. So we can say, Meta class ABC defines that the class which is derived from it becomes an abstract class.
- Abstract Class can have abstract method and concrete methods.  
Abstract Class needs to be extended and concrete methods implemented.
- PVM can not create objects of an abstract class.

Ex- from abc import ABC, abstractmethod

```
class Father(ABC):
    pass
```

Abstract Method-

- An abstract method is a method whose action is ~~dictated~~ redefined in the child classes as per the requirement of the object.
- We can declare a method as abstract method by using @abstractmethod decorator.

Syntax- `@abstractmethod`

```
def method_name(self):
    pass
```

Page No. 390  
Date

Ex- from abc import ABC, abstractmethod  
 class Father(ABC):  
 @abstractmethod  
 def disp(self):  
 pass.

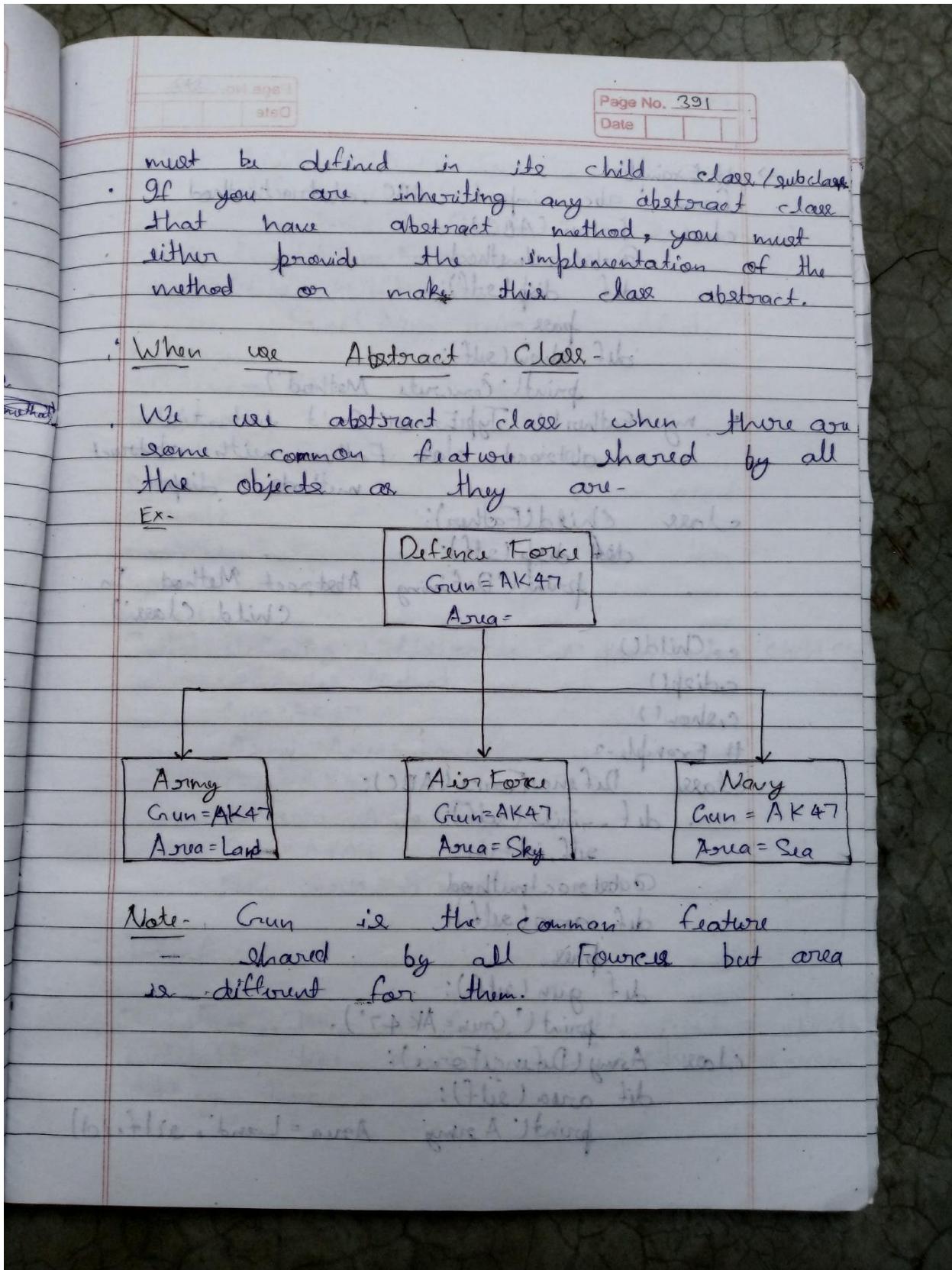
Concrete Method - A Concrete method is a method whose action is defined in the abstract class itself. (It is Normal Method E.g.)

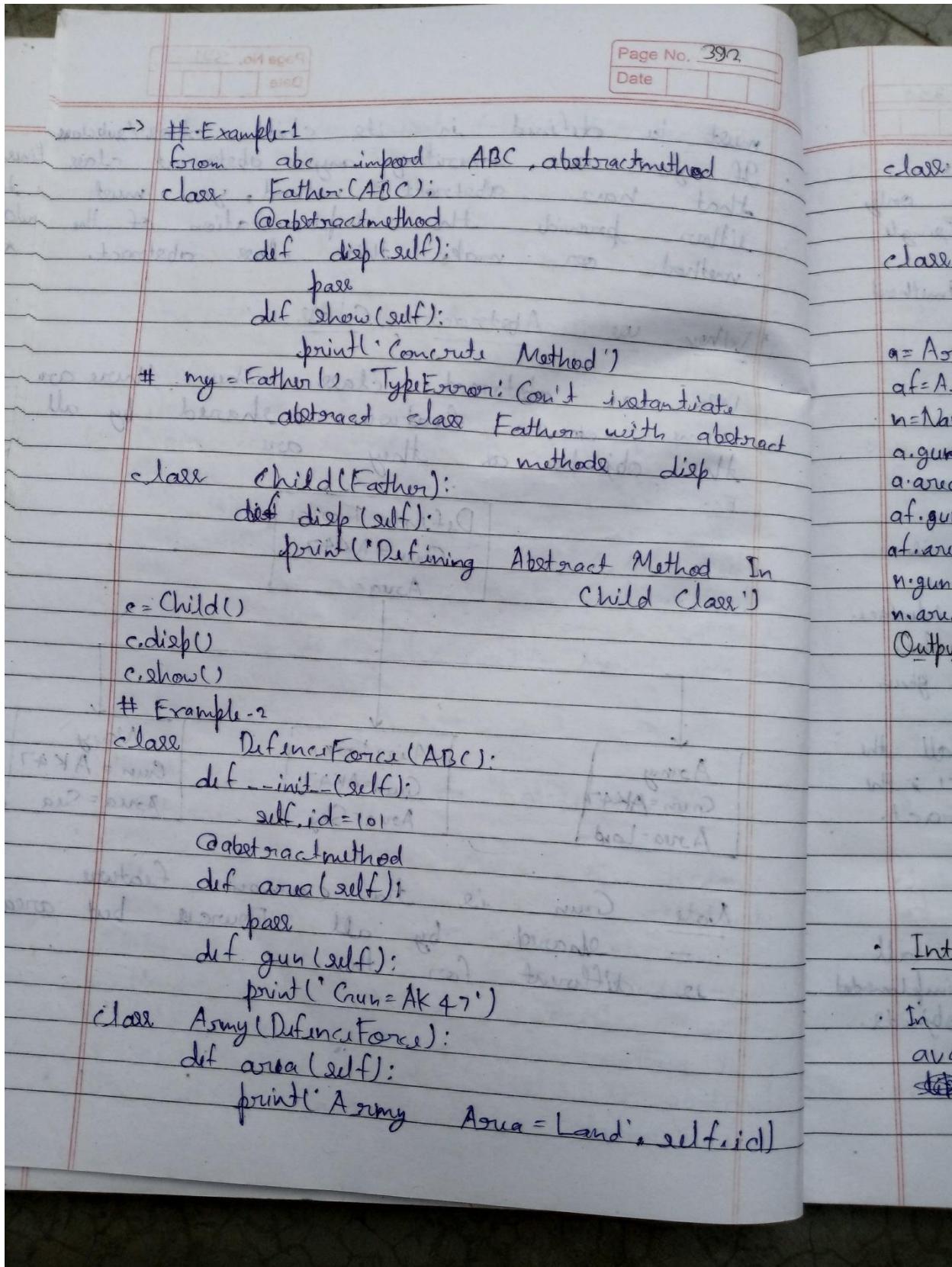
Ex- from abc import ABC, abstractmethod  
 @class Father(ABC):  
 @abstractmethod  
 def disp(self):  
 pass. } Abstract Method / Method without Body  
 def show(self):  
 print('Concrete Method') } Concrete Method / Method with Body

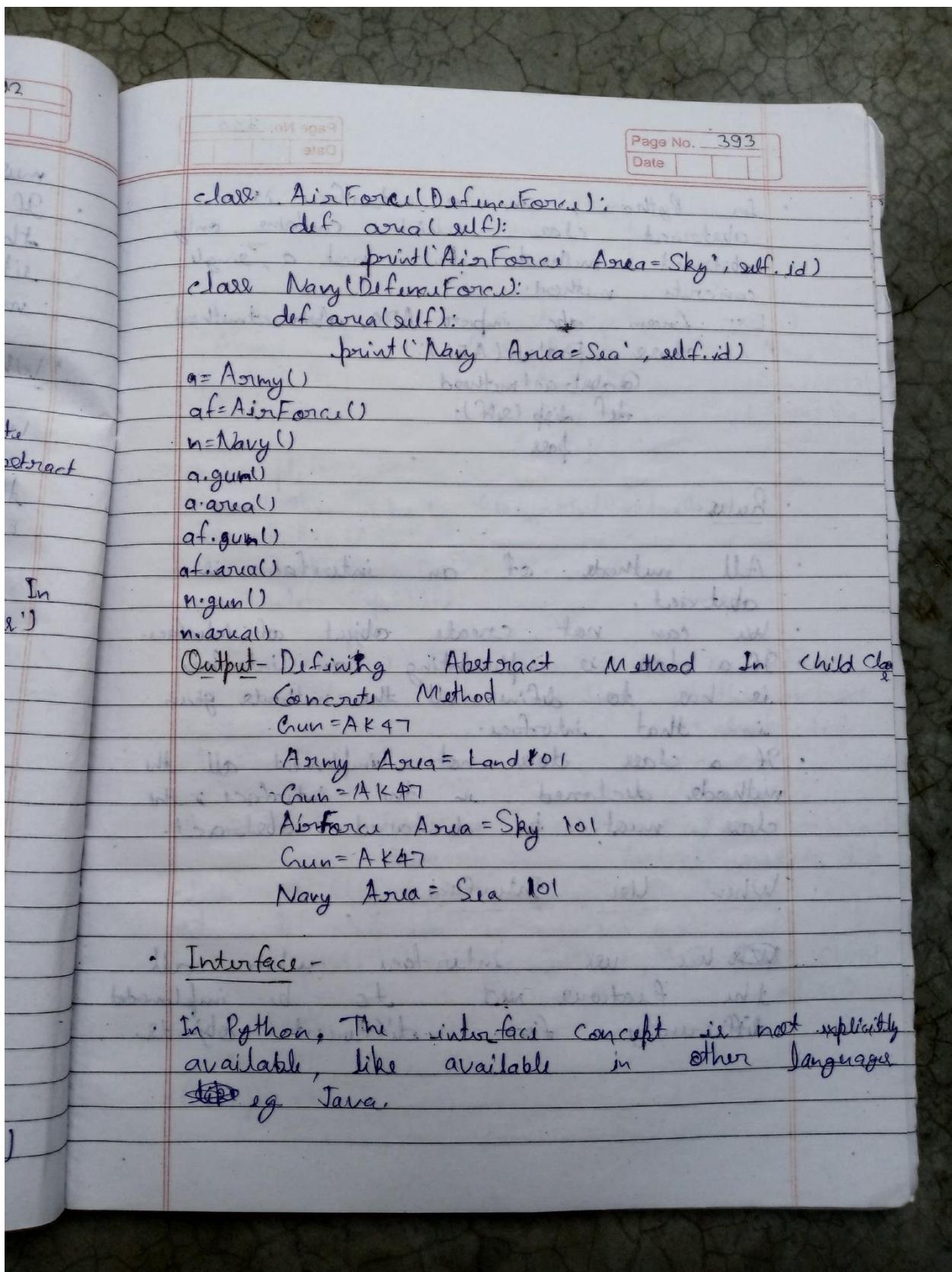
Rules-

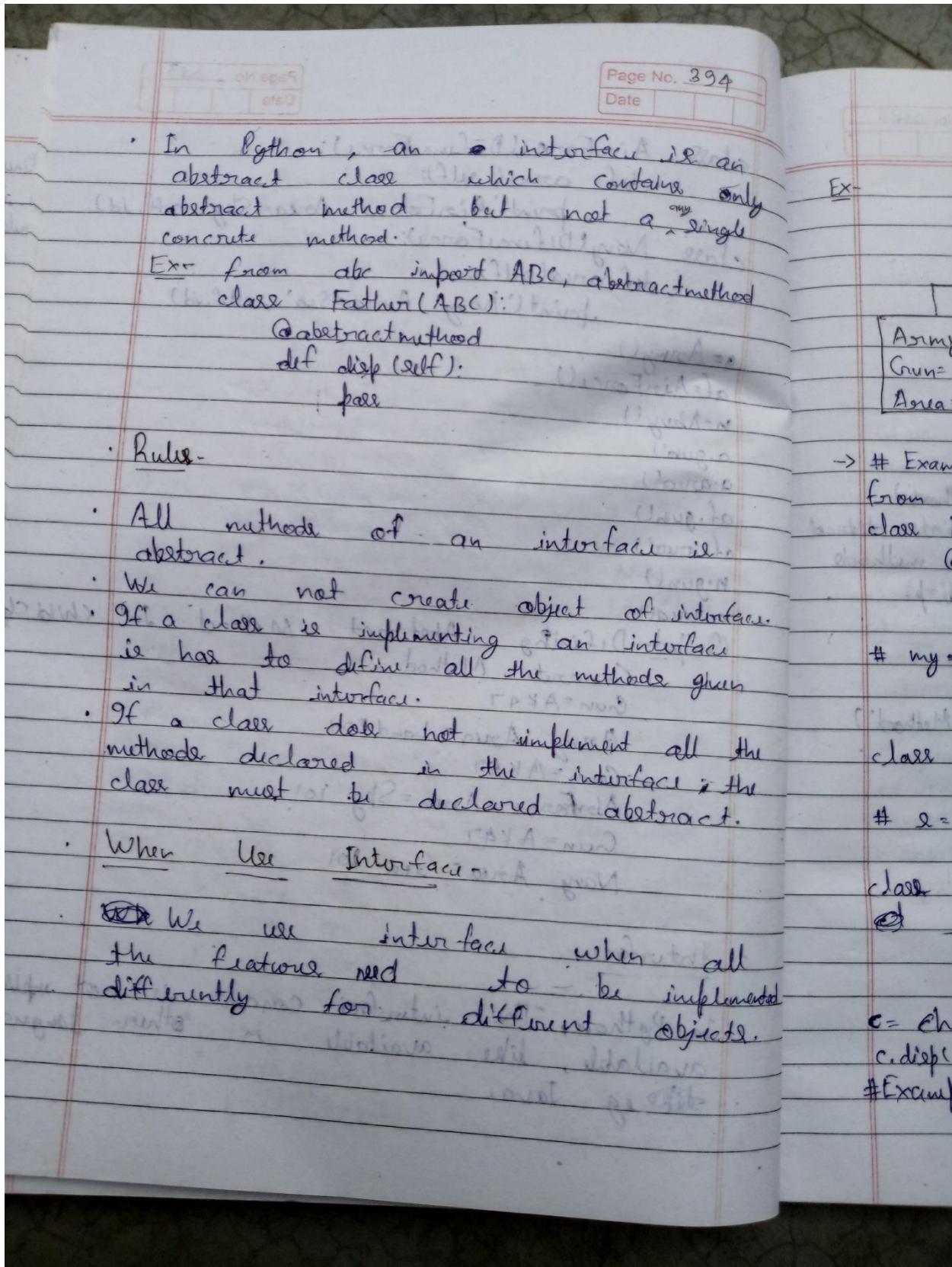
- PVM can not create objects of an abstract class.
- It is not necessary to declare all methods abstract in a abstract class.
- Abstract class can have abstract method and concrete methods.
- If there is any abstract method in a class, that class must be abstract.
- The abstract methods of a abstract class

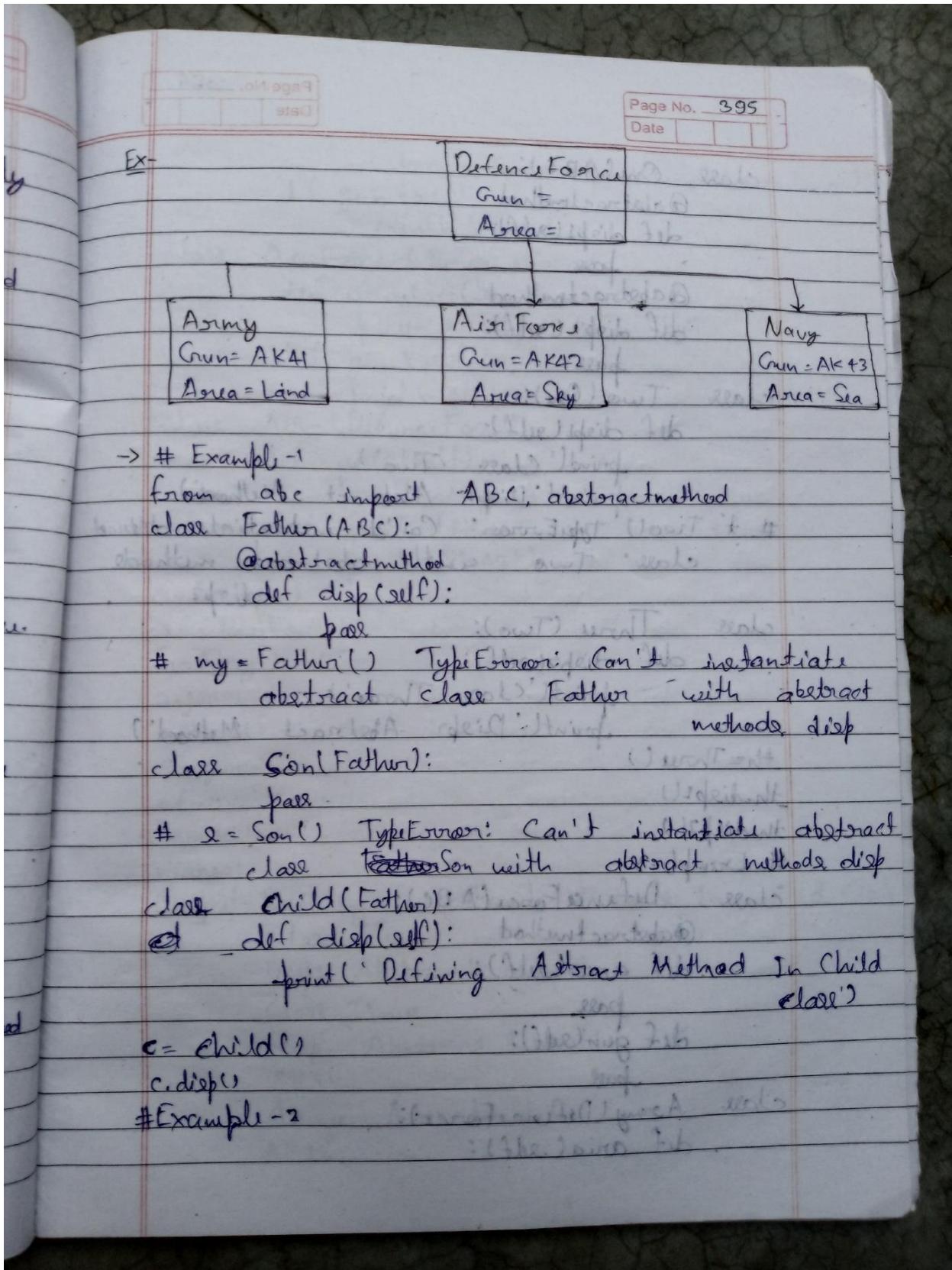
must  
If  
that  
either  
method  
  
When  
We  
same  
the  
Ex-  
  
Arming  
Gru...  
Area  
  
Note-  
se  
Uninten

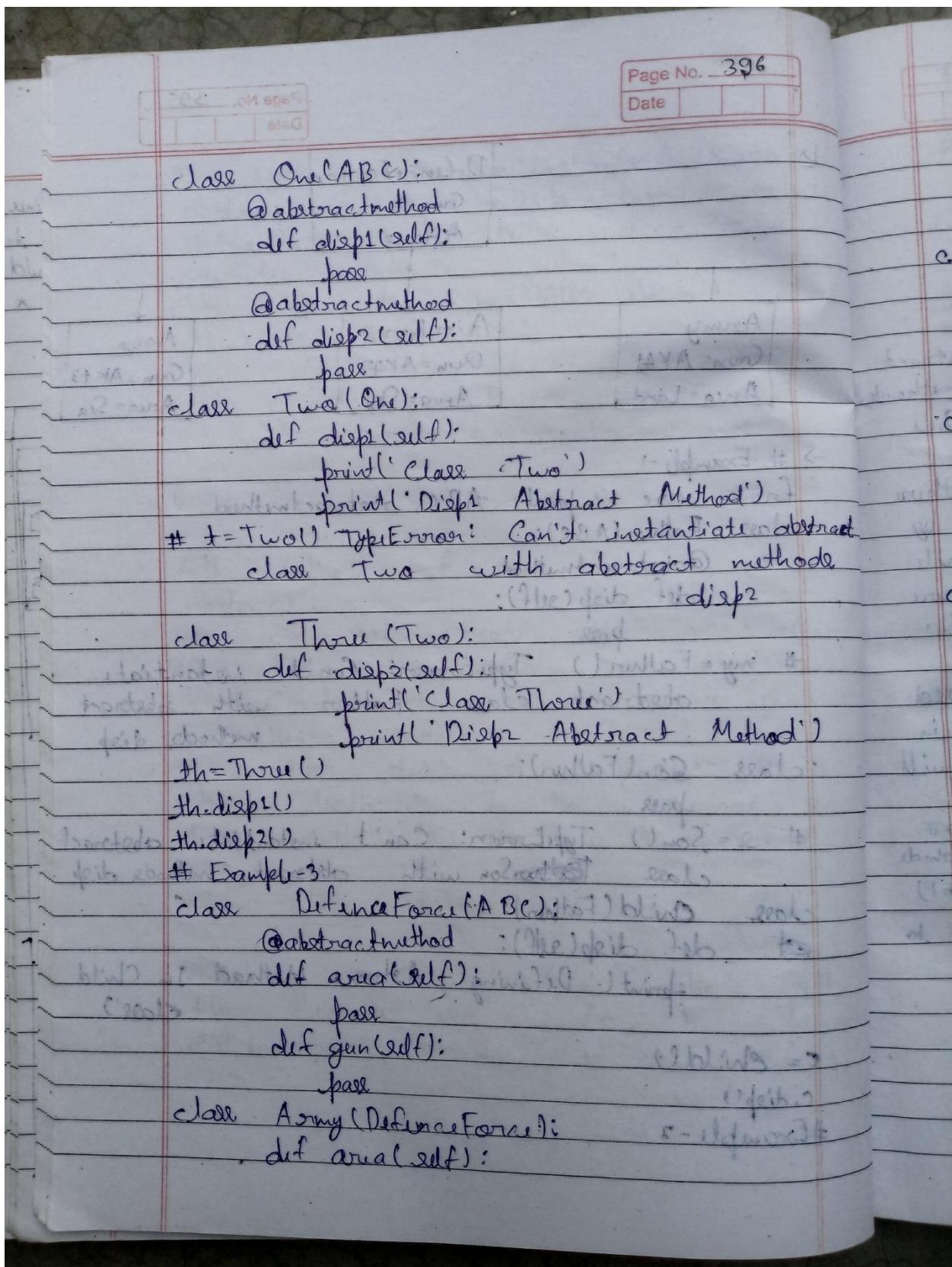


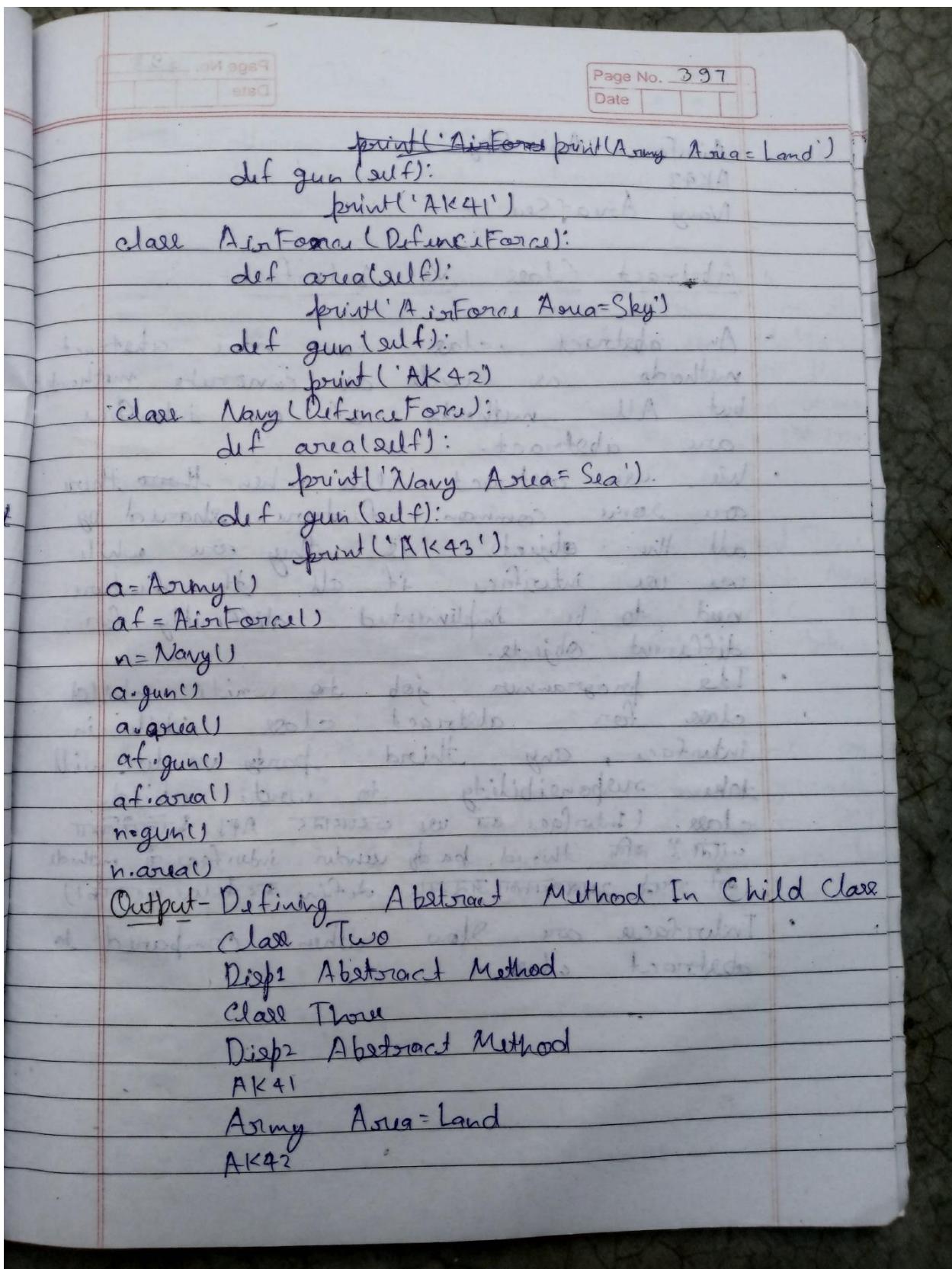


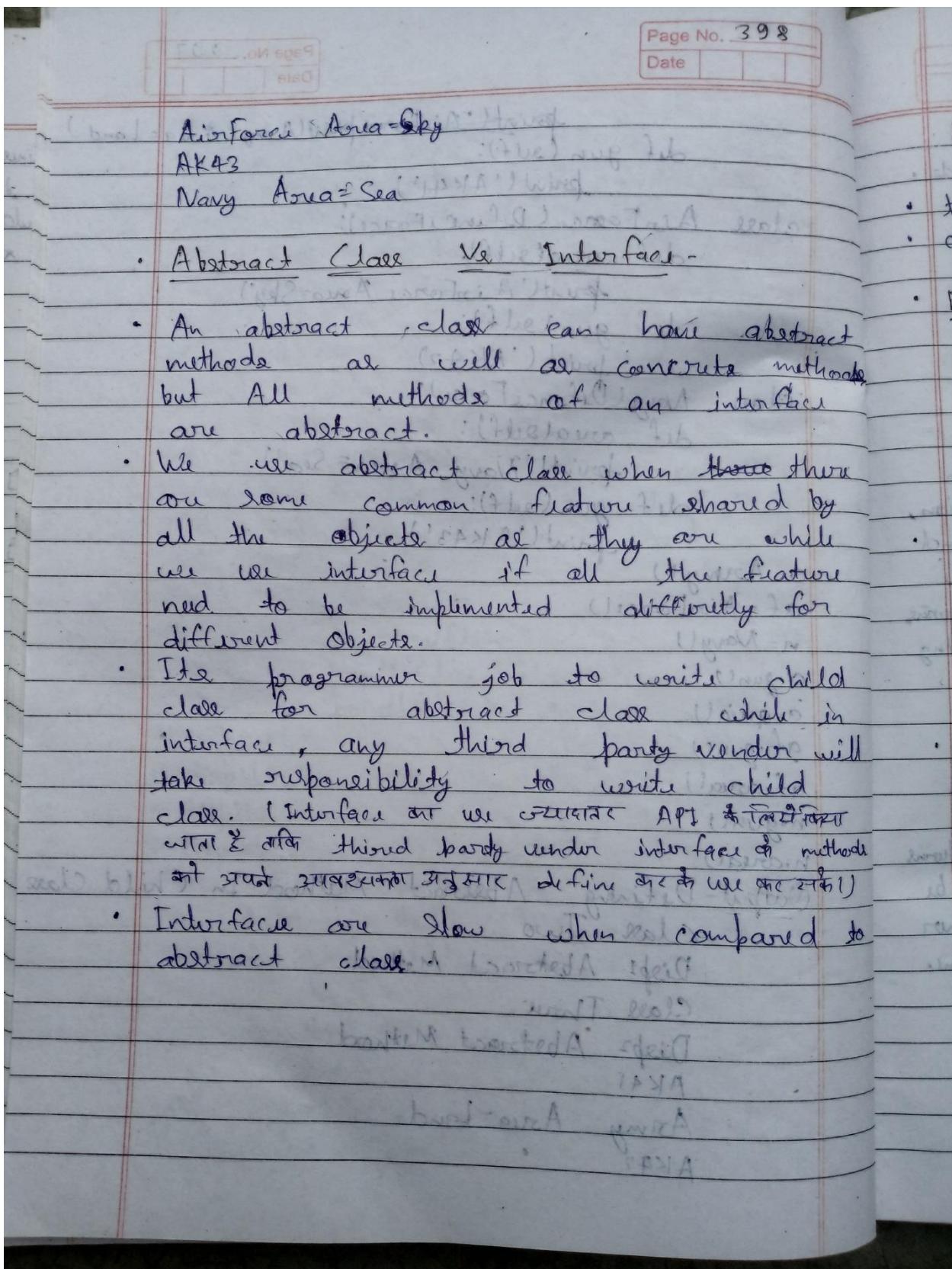


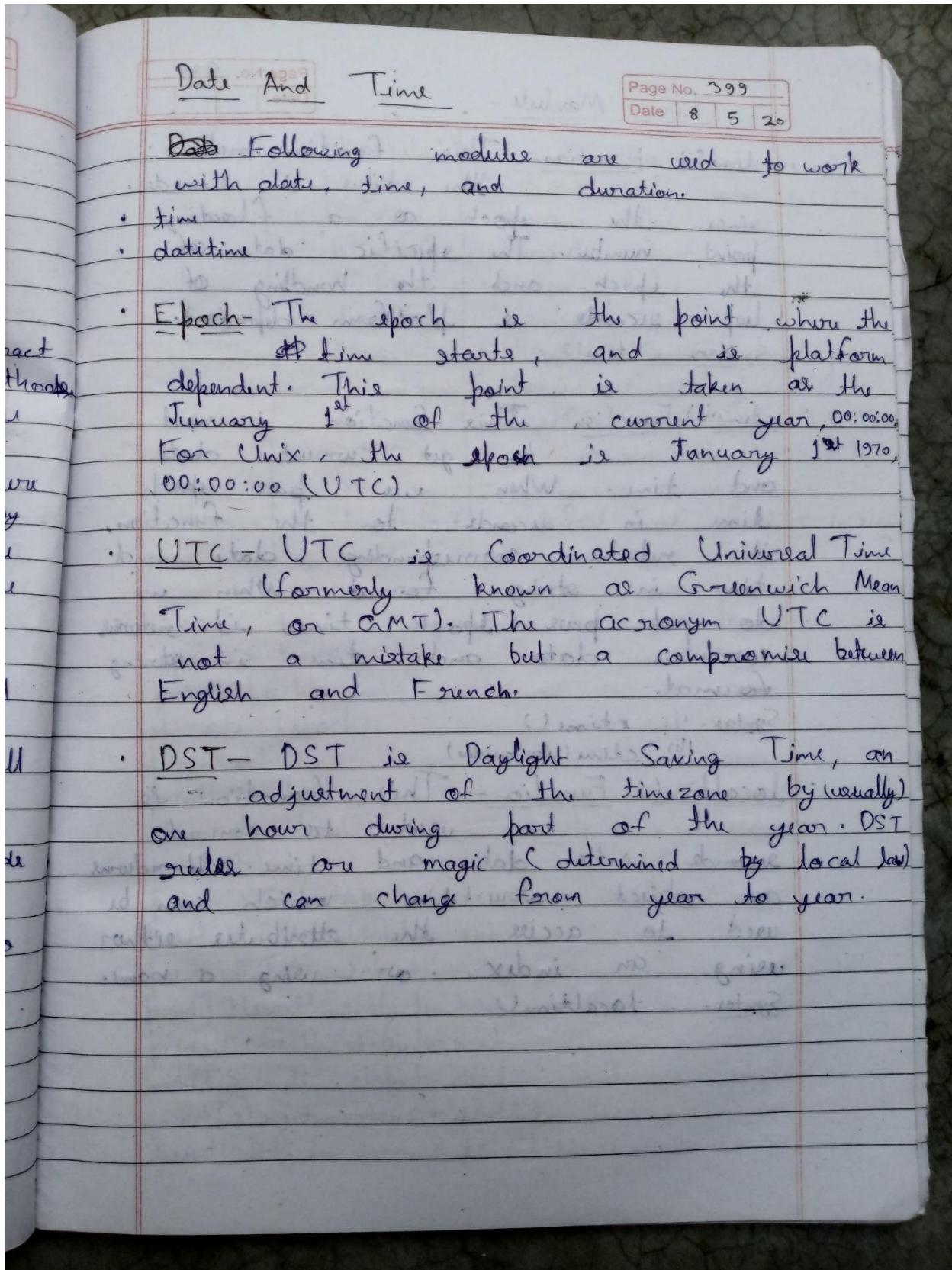


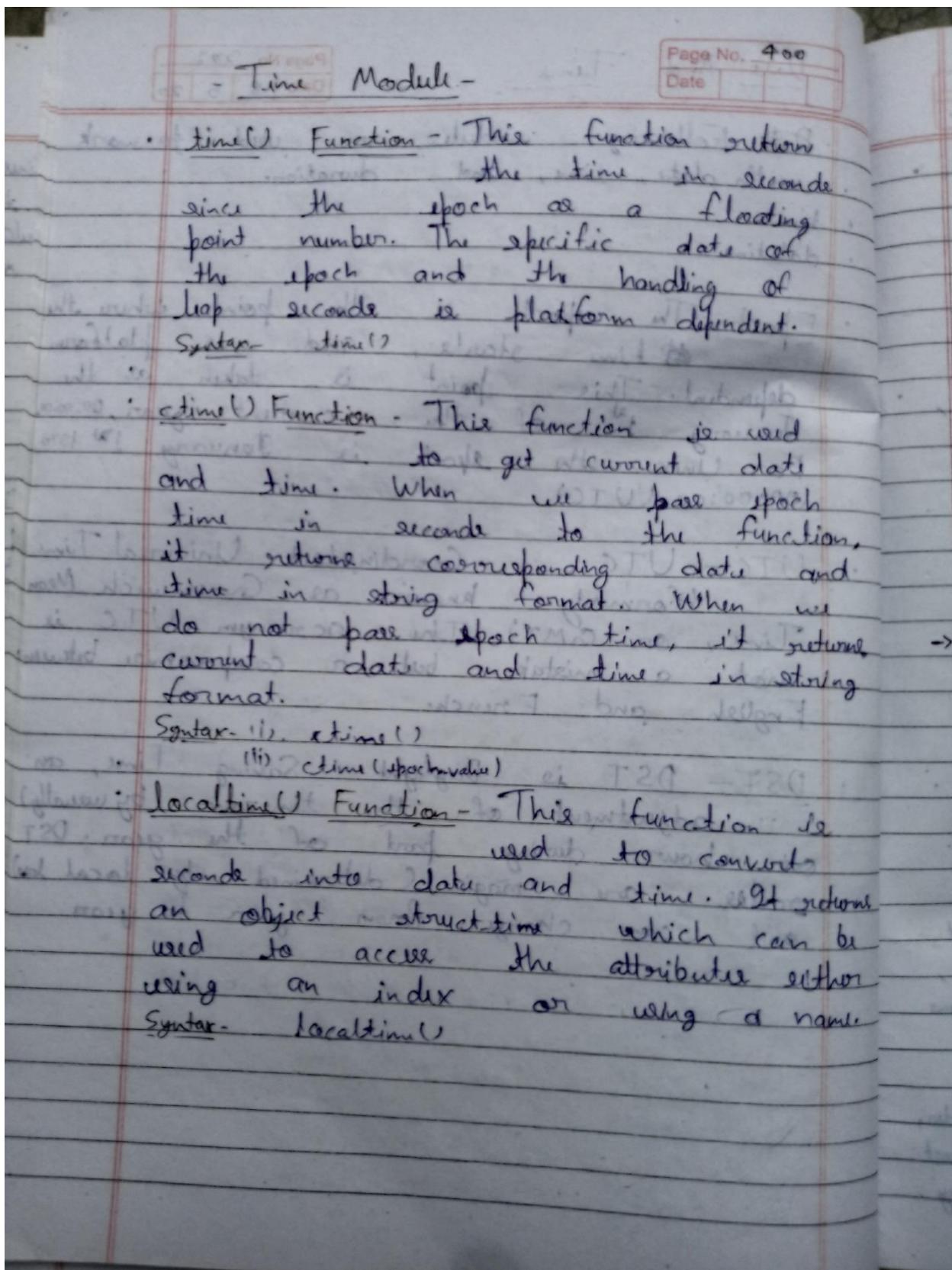






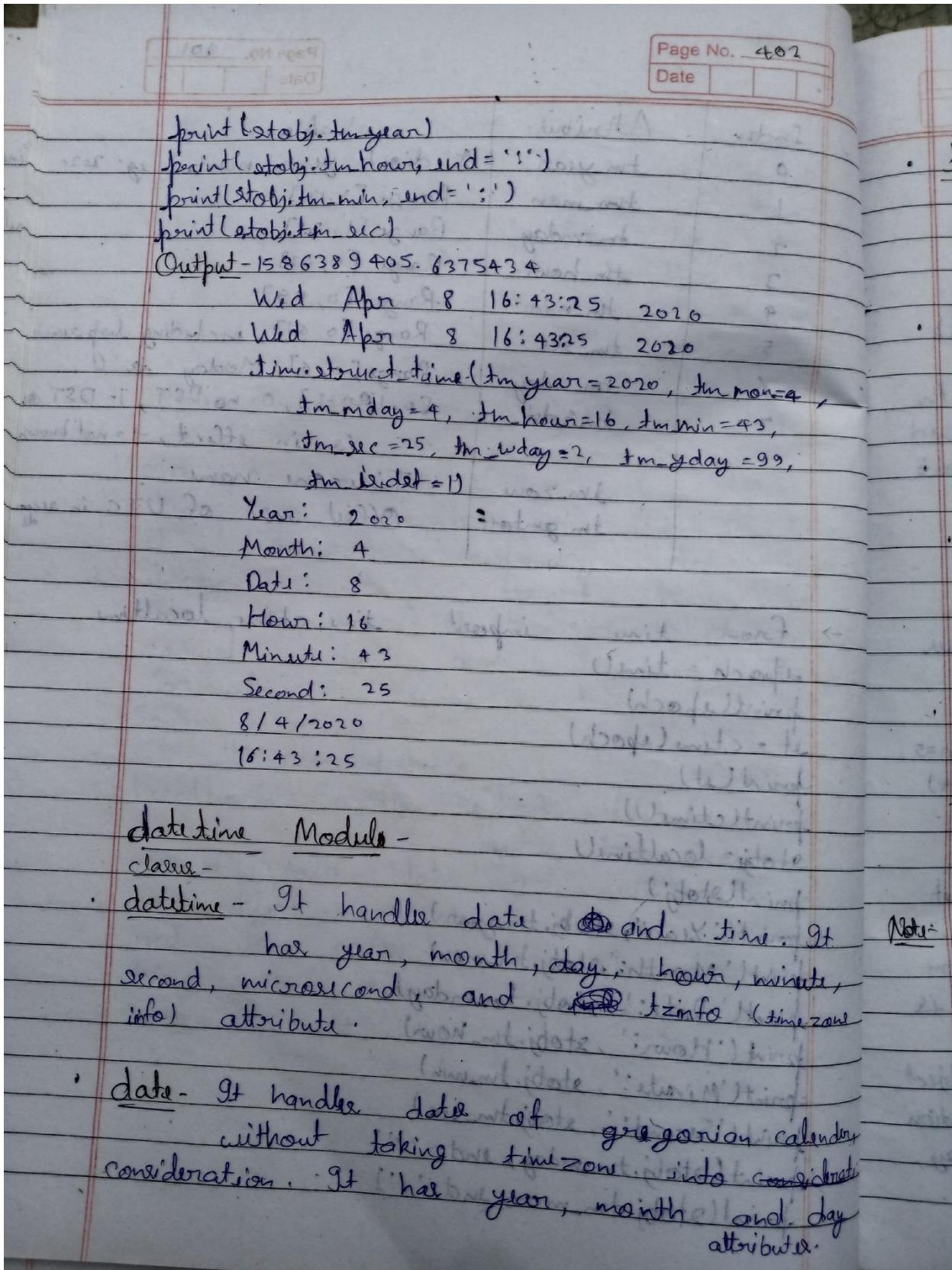


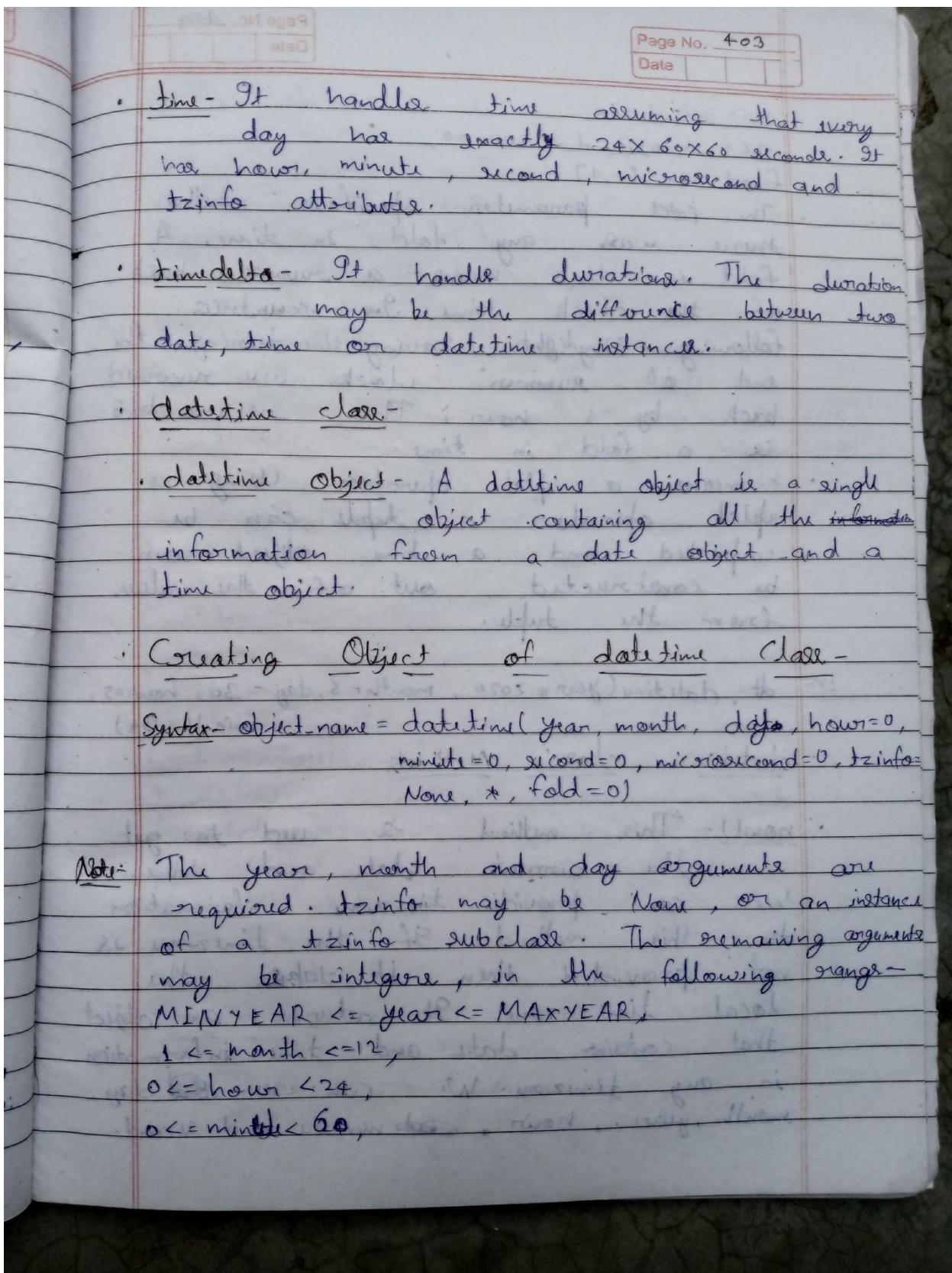




Index	Attribute	Value
0	tm_year	4 digit year number eg. 2020
1	tm_mon	Range [1, 12]
2	tm_mday	Range [1, 31]
3	tm_hour	Range [0, 23]
4	tm_min	Range [0, 59]
5	tm_sec	Range [0, 61], including leap seconds
6	tm_wday	Range [0, 6], Monday is 0
7	tm_isdst	[0, 1 or -1], 0= no DST, 1= DST is in effect, -1= not known
	tm_zone	Timezone name
	tm_gmtoff	Offset east of UTC in seconds

→ from time import time, ctime, localtime  
 epoch = time()  
 print(epoch)  
 st = ctime(epoch)  
 print(st)  
 print(ctime())  
 stobj = localtime()  
 print(stobj)  
 print('Year:', stobj.tm\_year)  
 print('Month:', stobj.tm\_mon)  
 print('Date:', stobj.tm\_mday)  
 print('Hour:', stobj.tm\_hour)  
 print('Minute:', stobj.tm\_min)  
 print('Second:', stobj.tm\_sec)  
 print(stobj.tm\_isdst, end=' ')  
 print(stobj.tm\_gmtoff, end=' ')





Page No. 404  
Date

$0 \leq \text{second} < 60$ ,  
 $0 \leq \text{microsecond} < 1000000$ ,

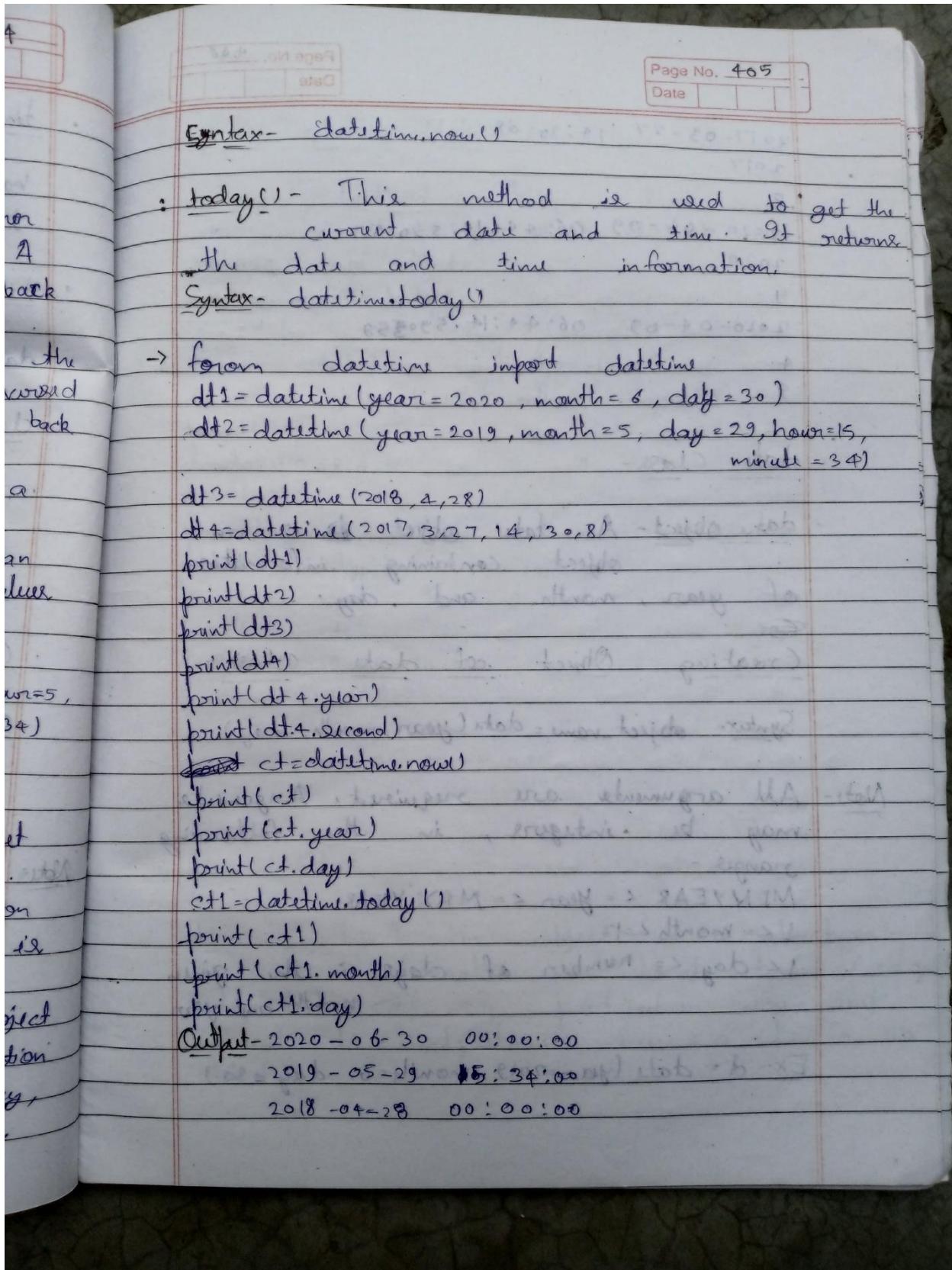
fold in [0, 1]

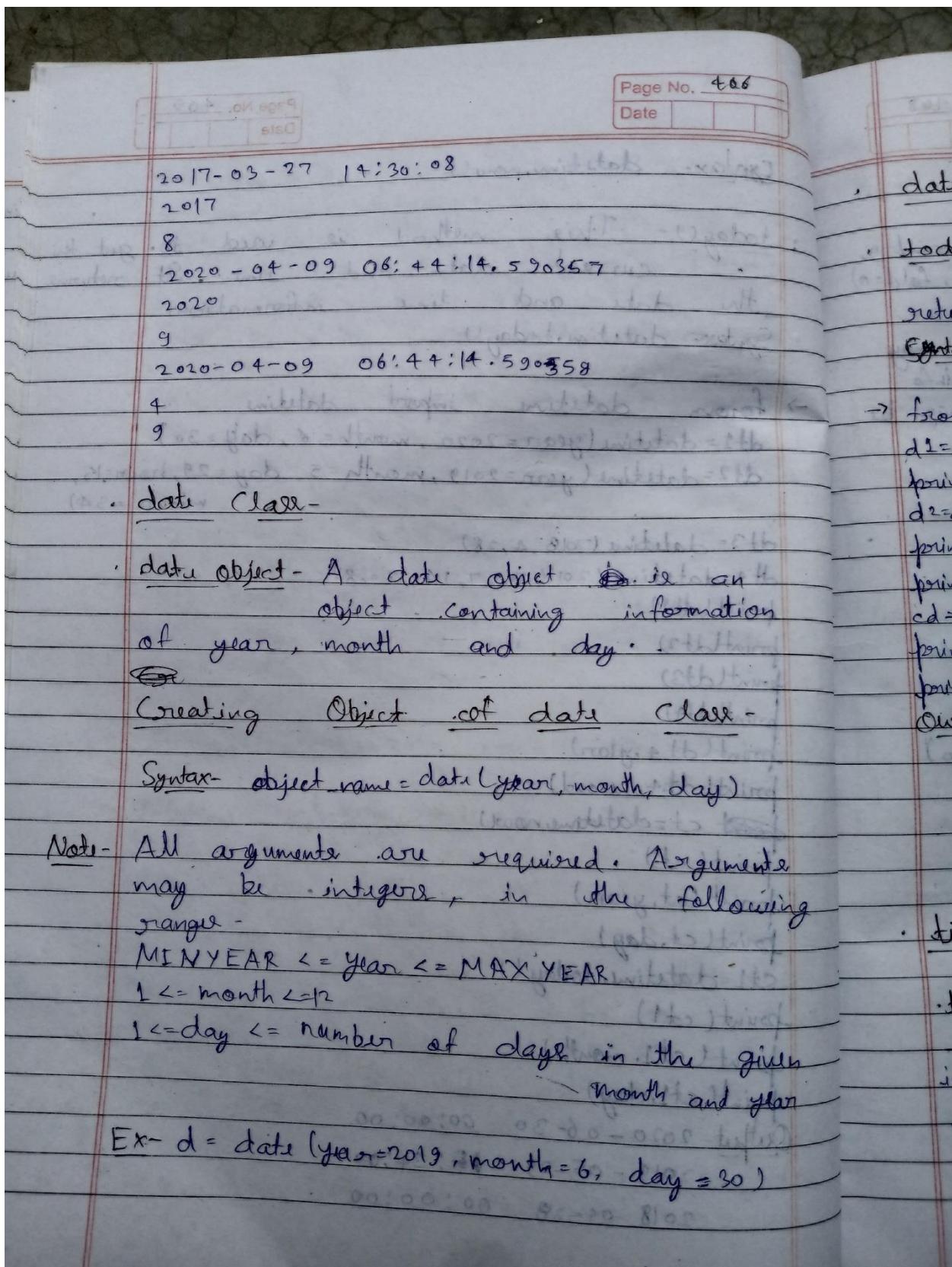
- The fold parameter specifies whether there was any fold in time. A fold in time means a reverse back of the clock time. In countries following Daylight saving time during the end of summer clocks are reversed back by 1 hour. This reverse back is a fold in time.
- \* means a splat operator. Using a splat operator a tuple can be unpacked and a time object can be constructed out of the values from the tuple.

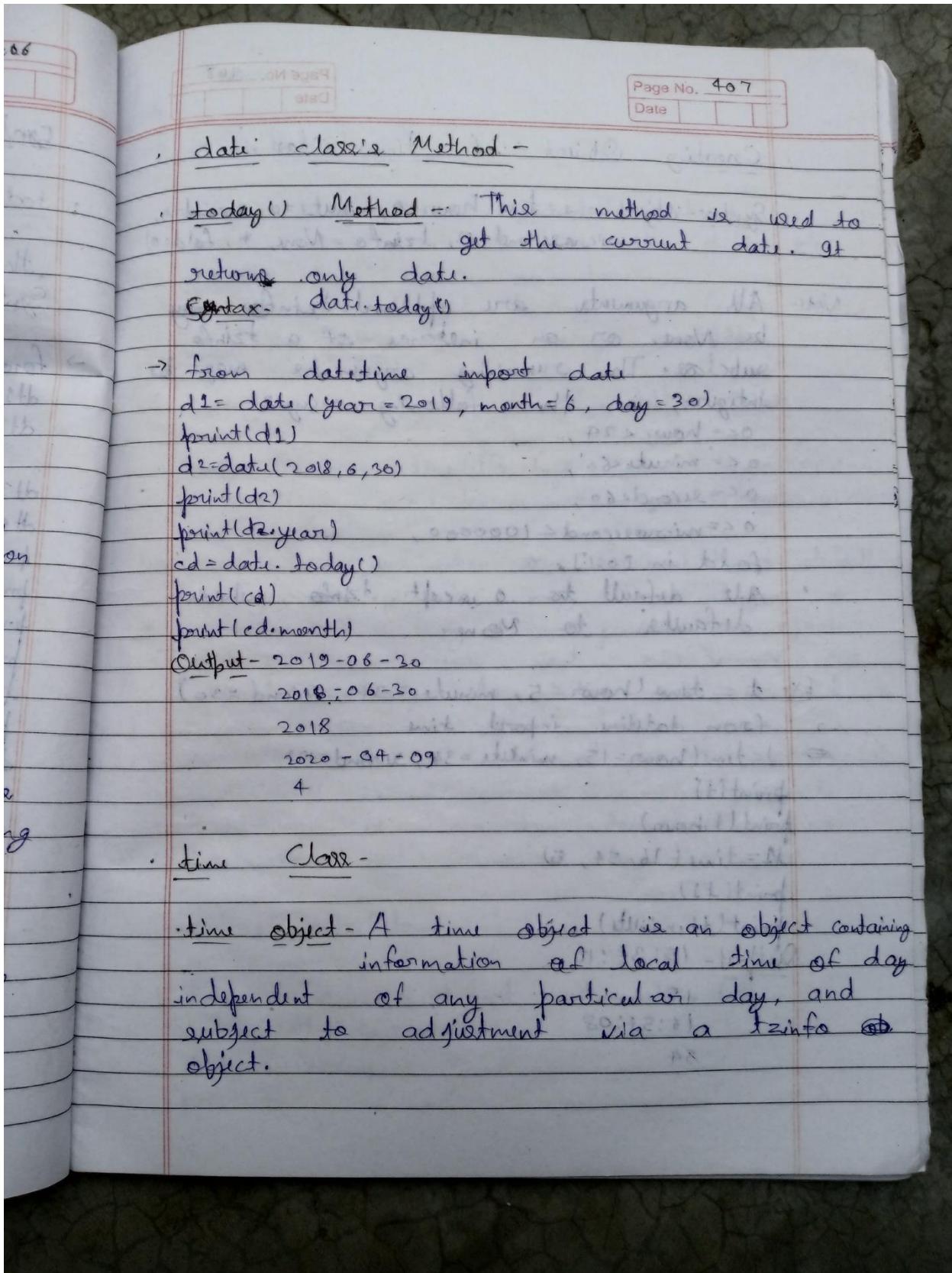
Ex: `dt = datetime(year=2020, month=6, day=30, hour=5, minute=34)`

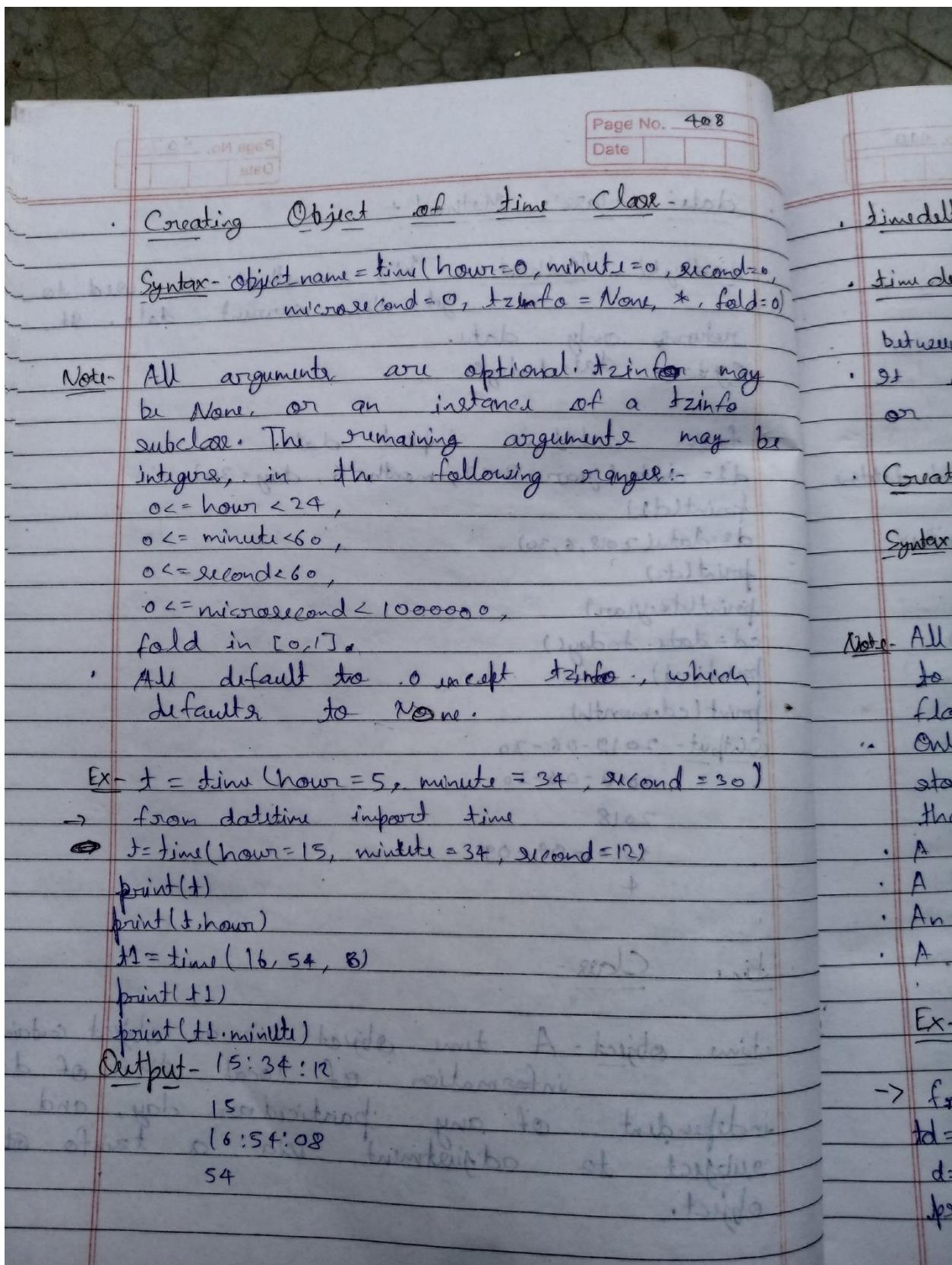
datetime class's Methods:

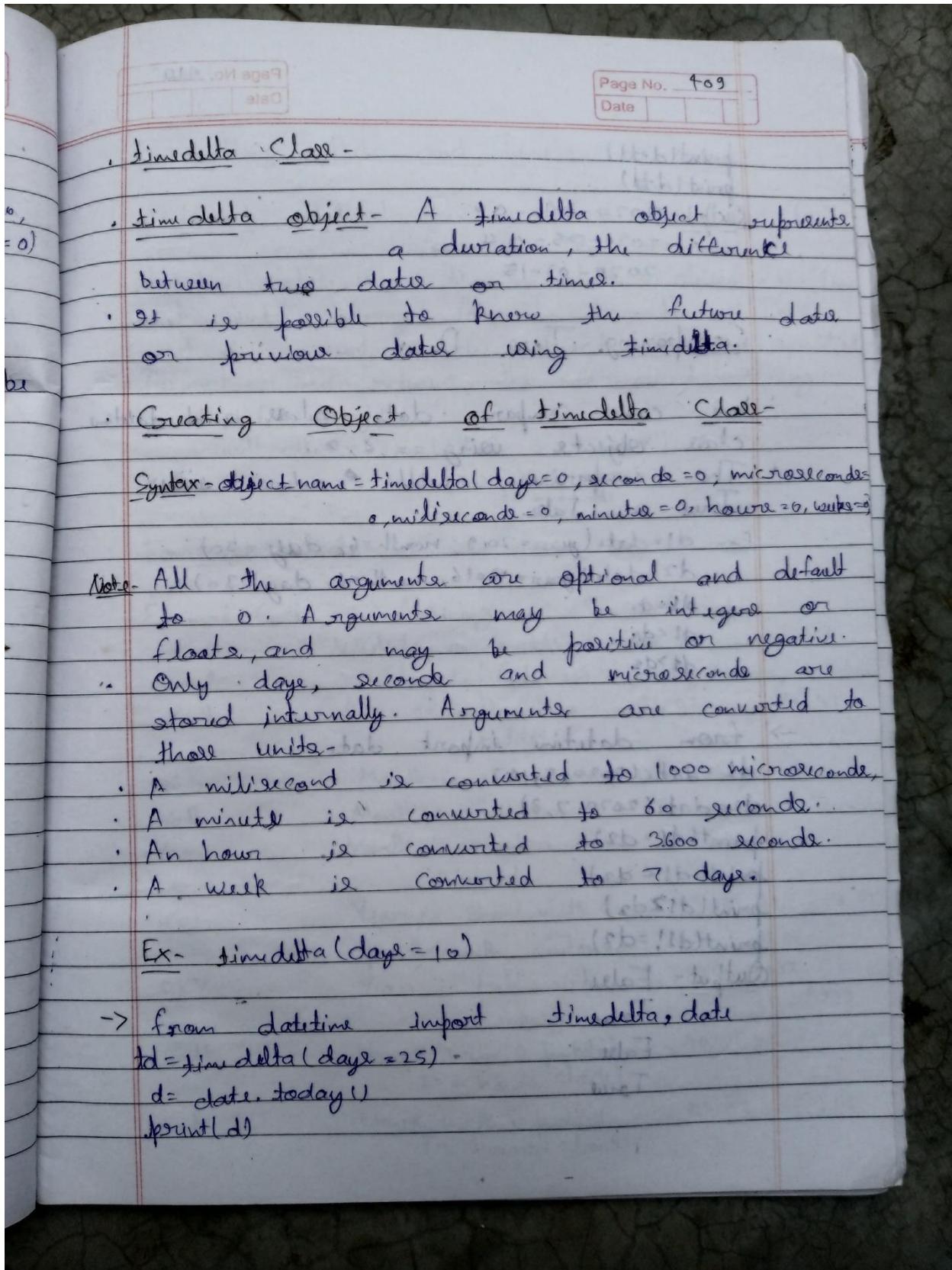
- now() - This method is used to get the current date and time. We can provide timezone information to this method. If the timezone is not provided, then it takes the local time zone. It returns an object that contains date and time information in any timezone. We can use ~~day, month, year, hour, minutes and second~~.

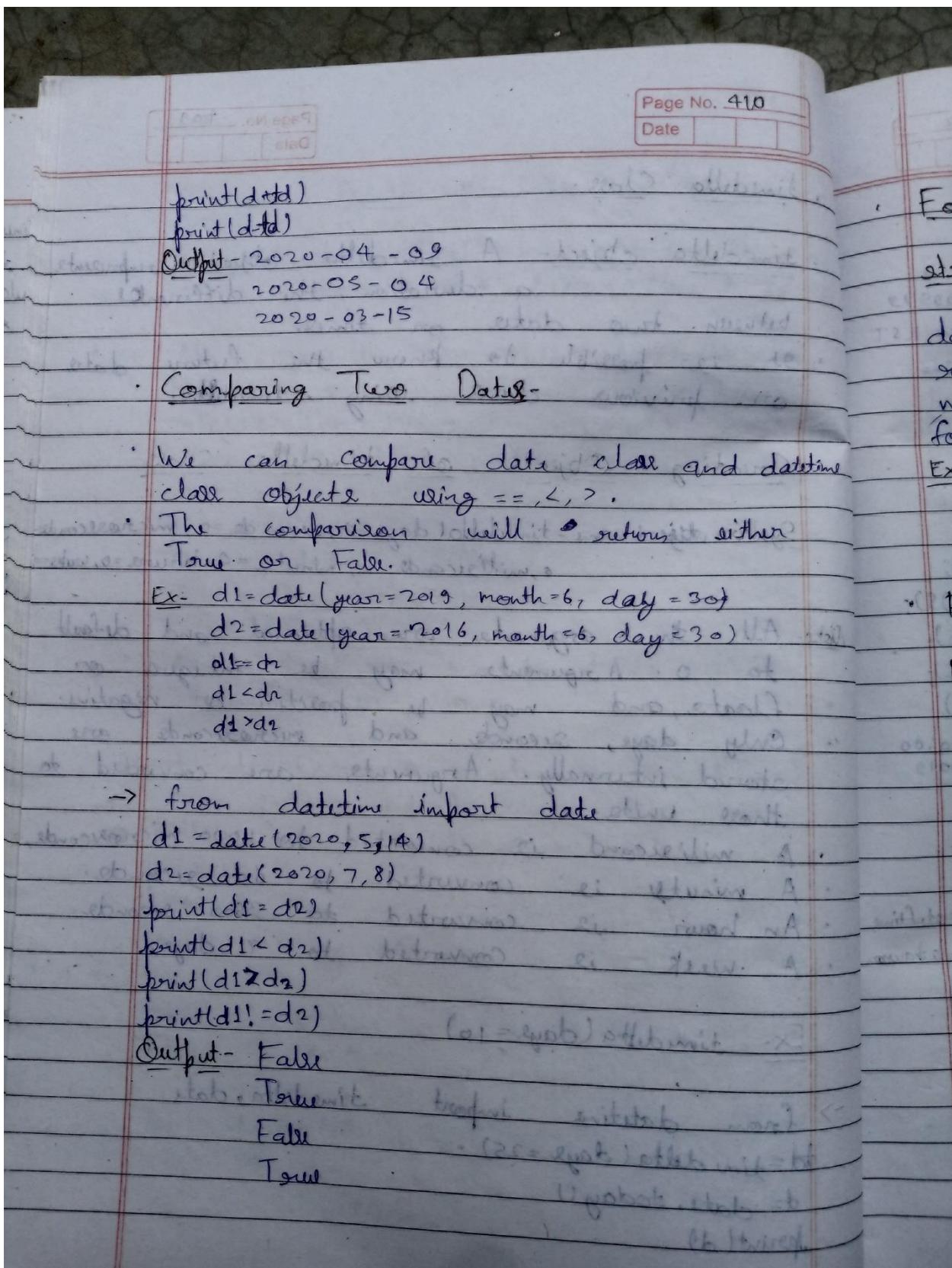


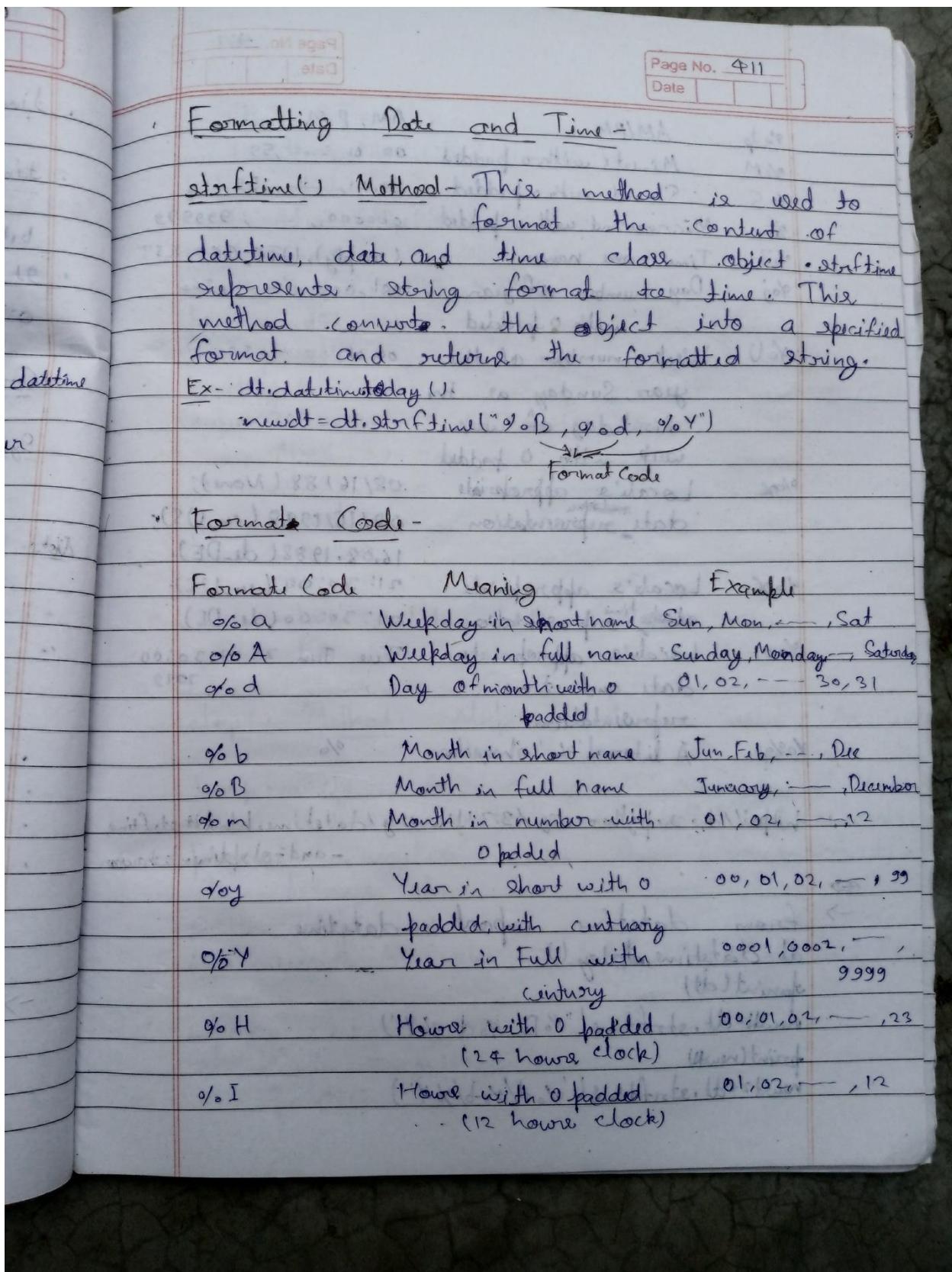




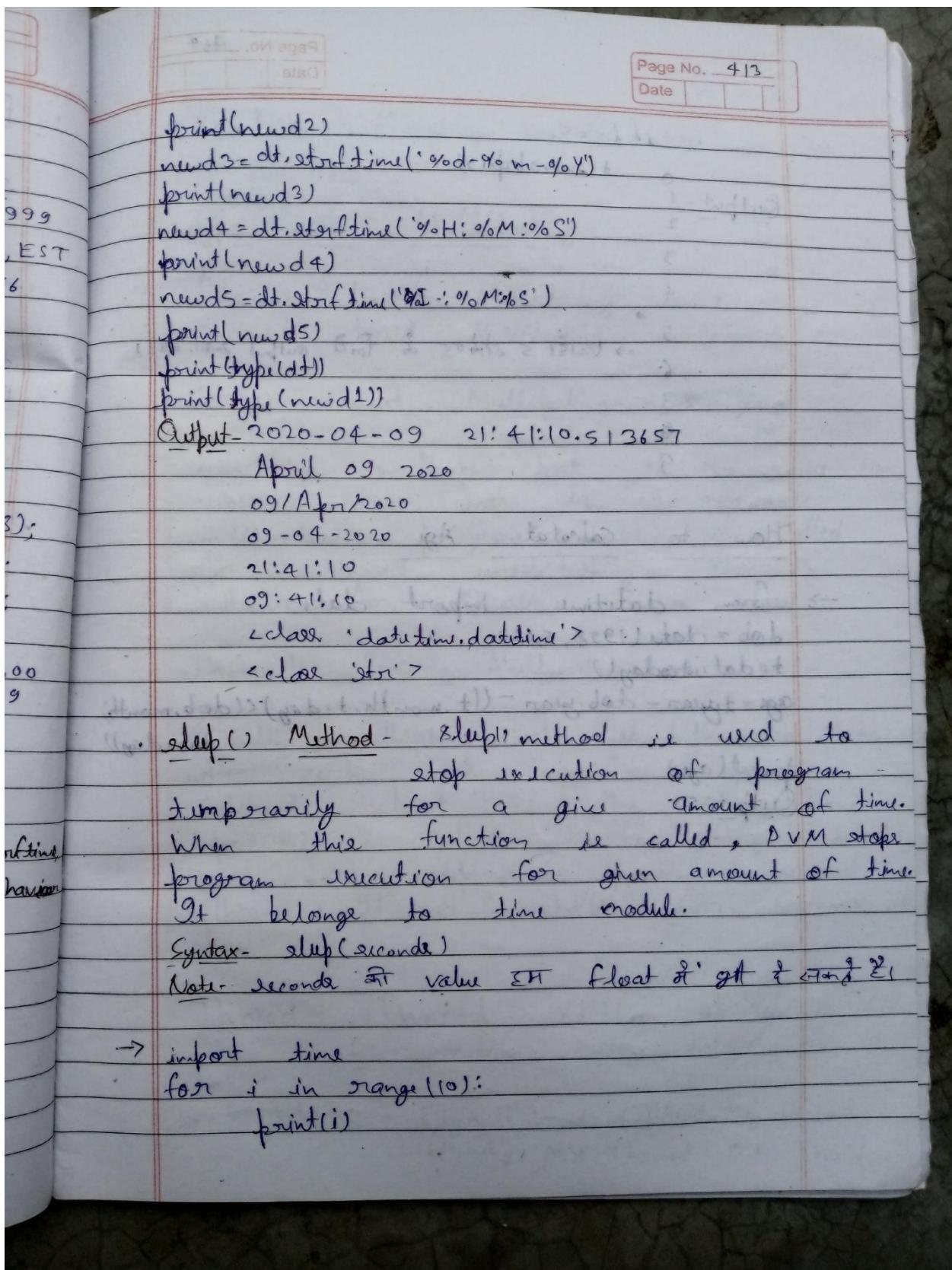


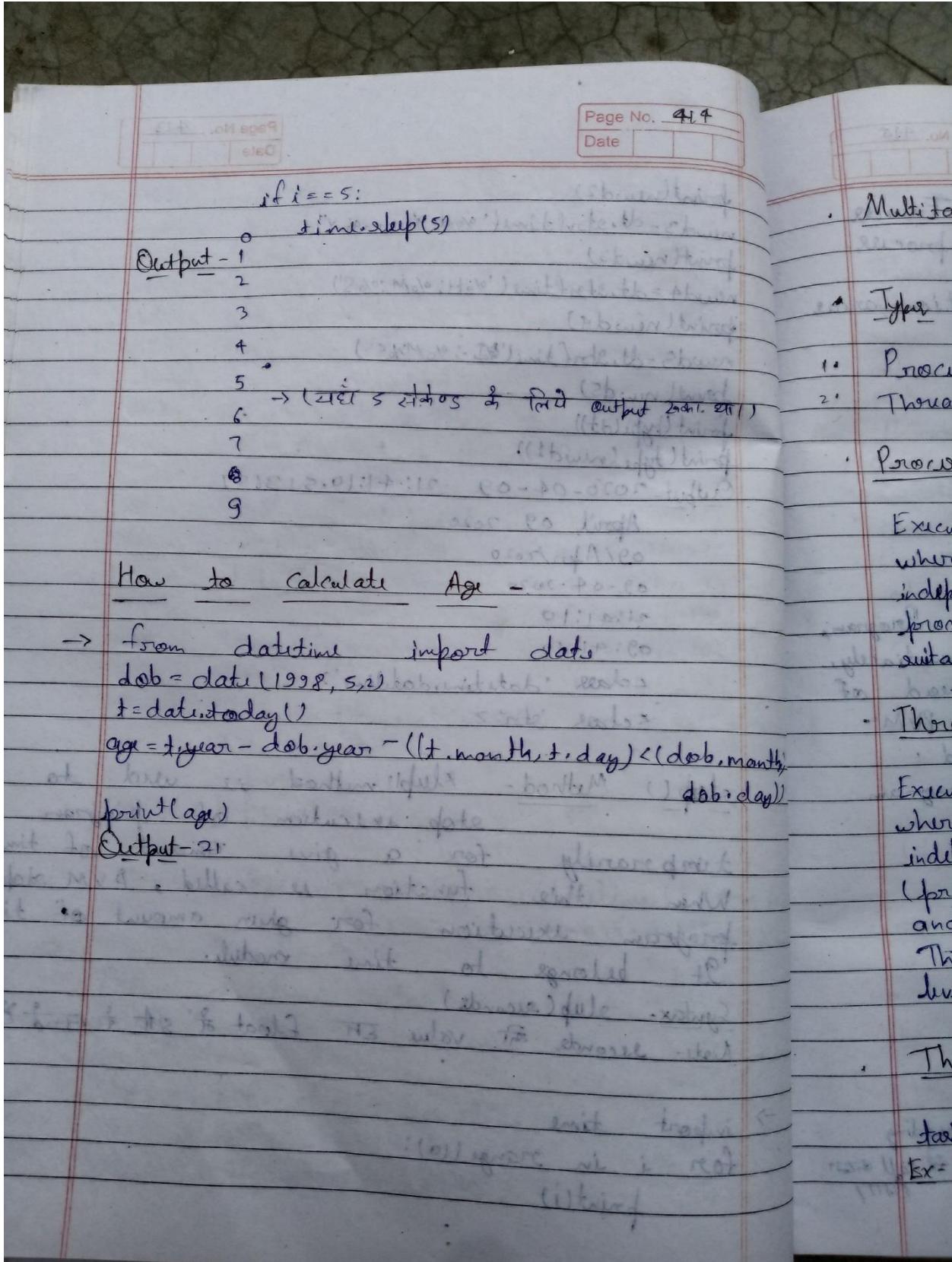


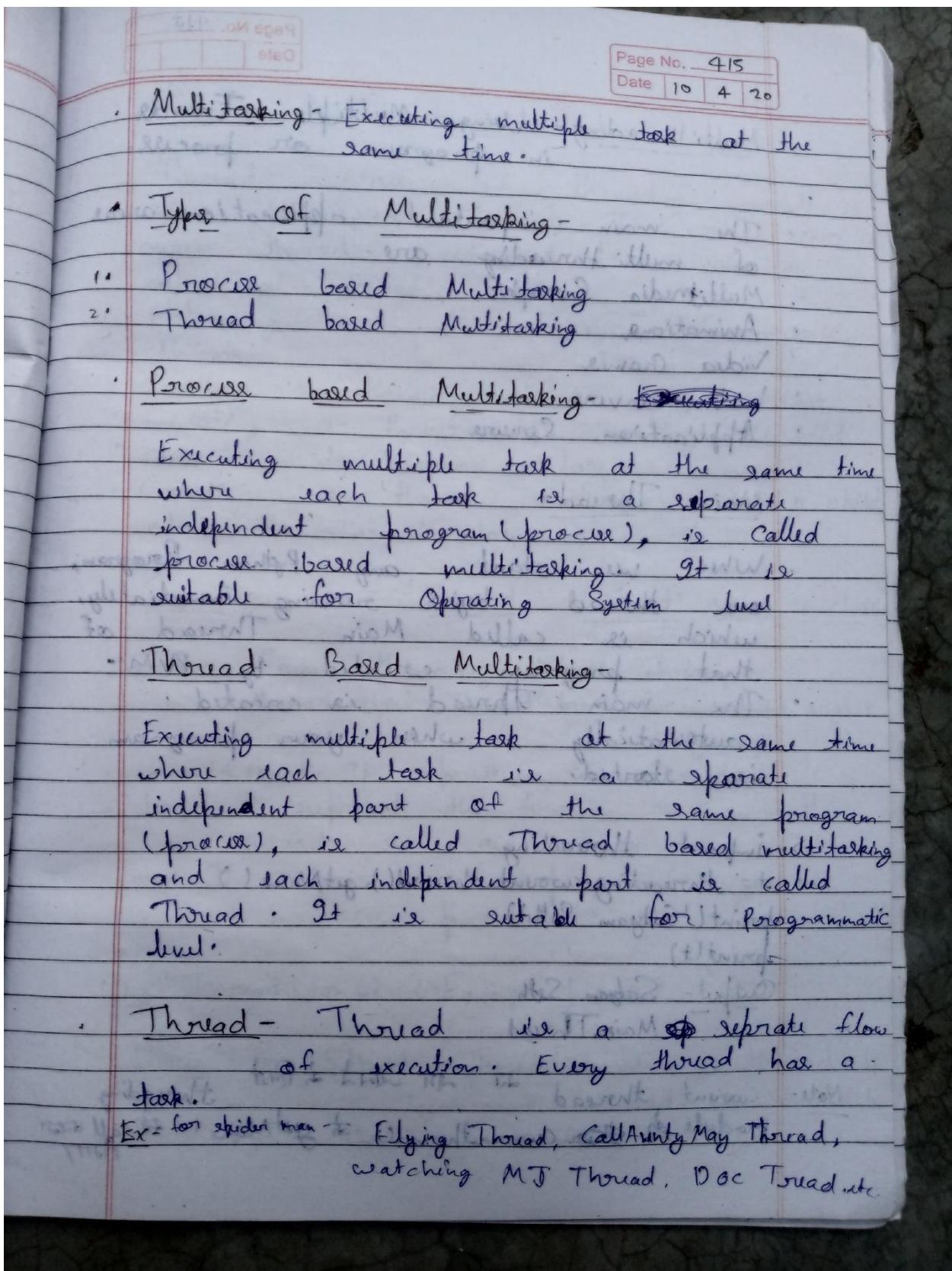




		Page No. 412	Date
%p	AM/PM	AM, PM	print newd
%M	Minute with 0 padded	00, 01, --, 59	print newd
%S	Second with 0 padded	00, 01, --, 59	print newd
%f	Microsecond with 0 padded	000000, --, 999999	print newd
%Z	Time zone name (empty), UTC, CST, EST	(empty), UTC, CST, EST	print newd
%j	Day number of year with 0 padded	001, 002, --, 366	print newd
%U	Week number of the year, Sunday as the first day of week with 0 padded	00, 01, --, 53	print newd
%c	Locale's appropriate date representation	08/16/88 (None); 08/16/1988 (en-US); 16.08.1988 (de-DE).	print newd
%X	Locale's appropriate time representation	21:30:30 (en-US); 21:30:00 (de-DE).	print newd
%C	Locale's appropriate date and time representation	Tue Jun 30 21:30:00 2019	print newd
%%	A literal '%' character	%	print newd
<a href="https://docs.python.org/3.7/library/datetime.html#strftime-and-strptime-behavior">https://docs.python.org/3.7/library/datetime.html#strftime-and-strptime-behavior</a>			
→	from datetime import datetime dt = datetime.today() print(dt)	newd1 = dt.strftime("%B %d %Y") print(newd1) newd2 = dt.strftime("%d/%b/%Y") print(newd2)	→ for







Page No. 415  
Date:

- Multi-threading - Using Multiple Threads in program or process.
- The main important application areas of multi threading are -
  - Multimedia Graphics
  - Animations
  - Video Games
  - Web Servers
  - Application Servers
- Main Thread -
- When we start any Python Program, one thread begins running immediately, which is called Main Thread of that program created by PVM.
- The main thread is created automatically when your program is started.

→ `import threading  
t = threading.current_thread().getName()  
print('Satyam Seth')  
print(t)`

Output - Satyam Seth  
MainThread

Current thread is the current thread module of ~~currentthread()~~ threading at getName() it calls self

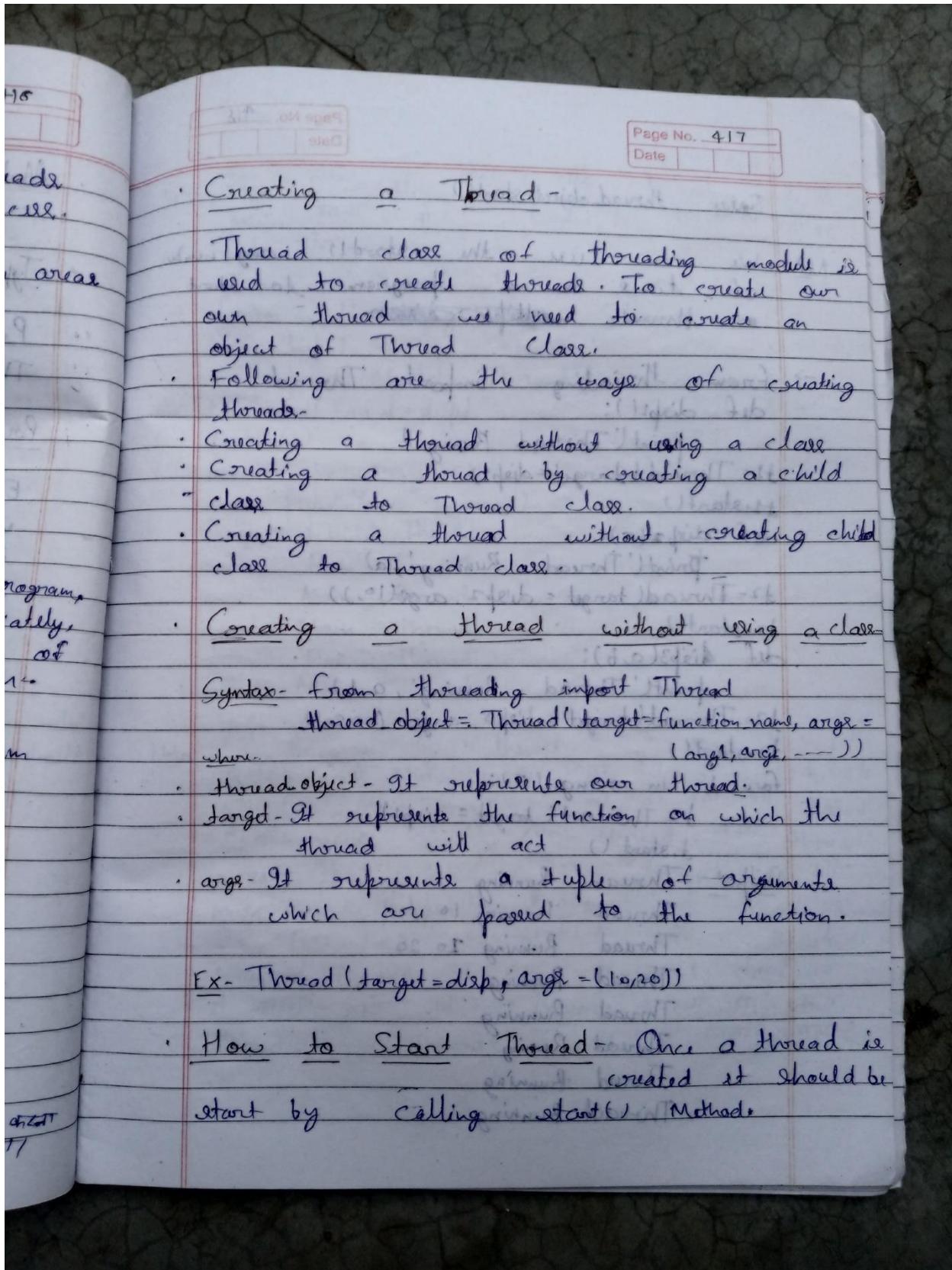
Creating Thread used to own the object of Following threads.

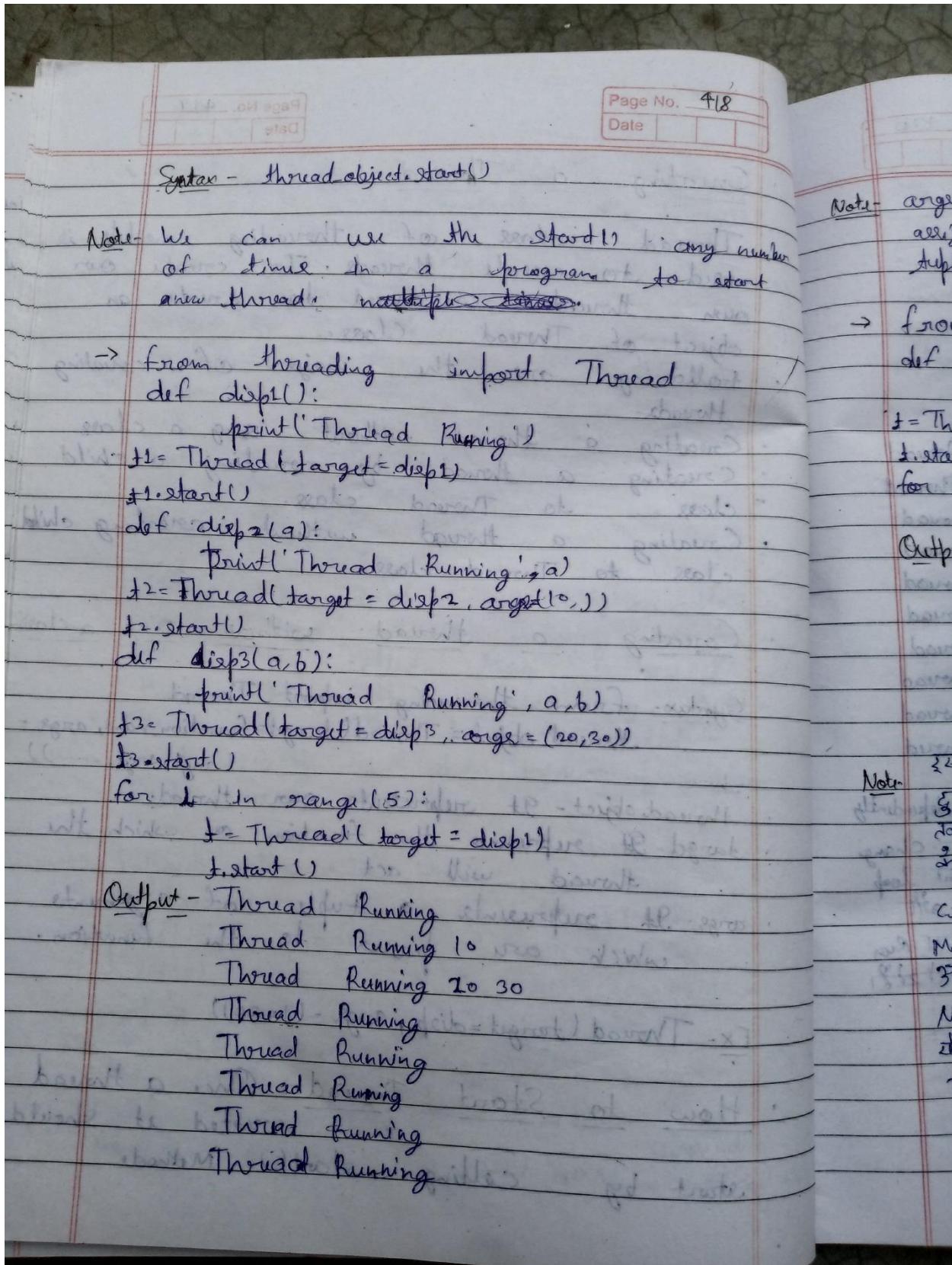
Creating Creating class Creating class

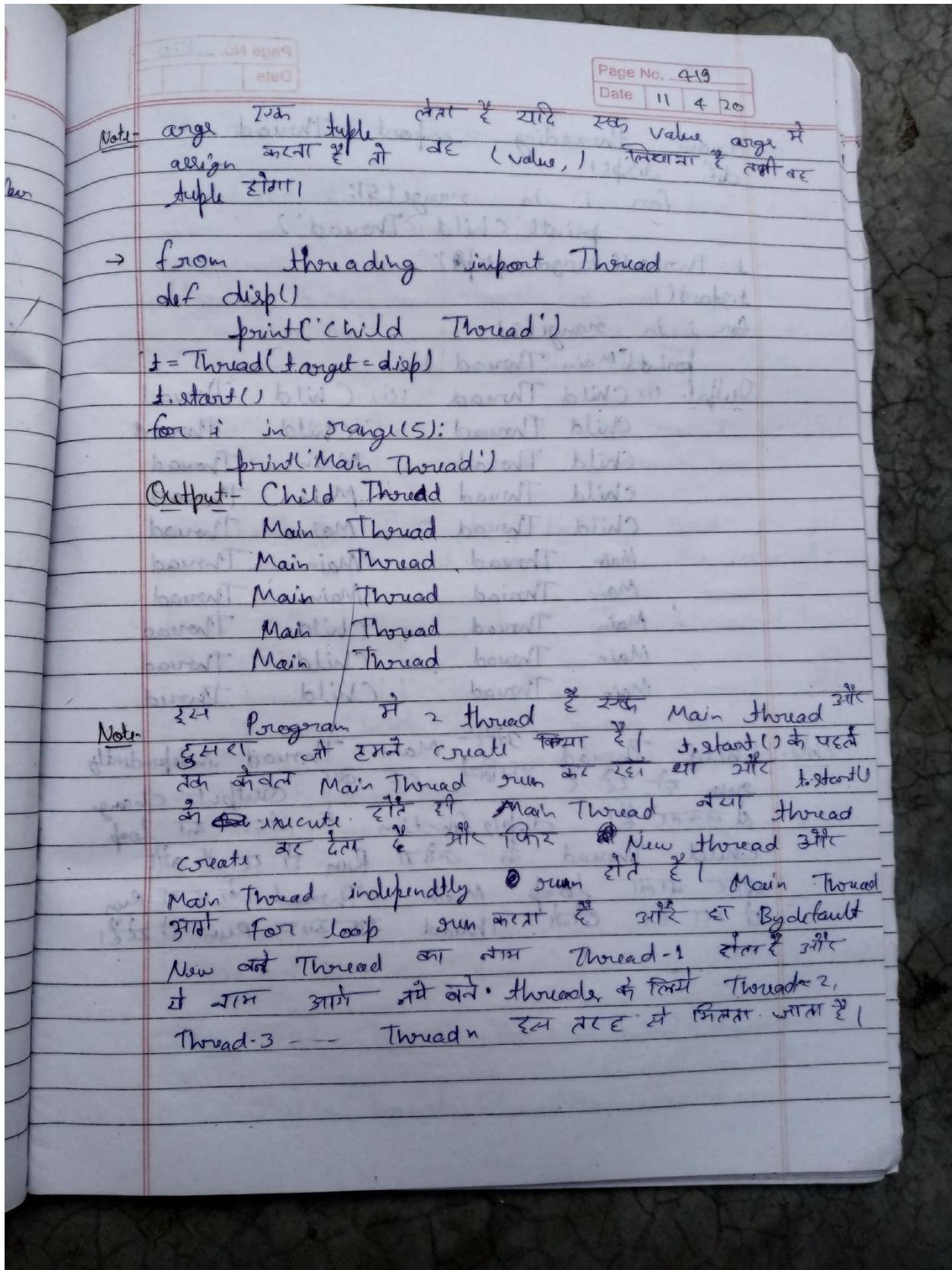
Creating Syntax - ~~the~~ where thread of ~~the~~ joined - ~~it~~ args - ~~it~~ who

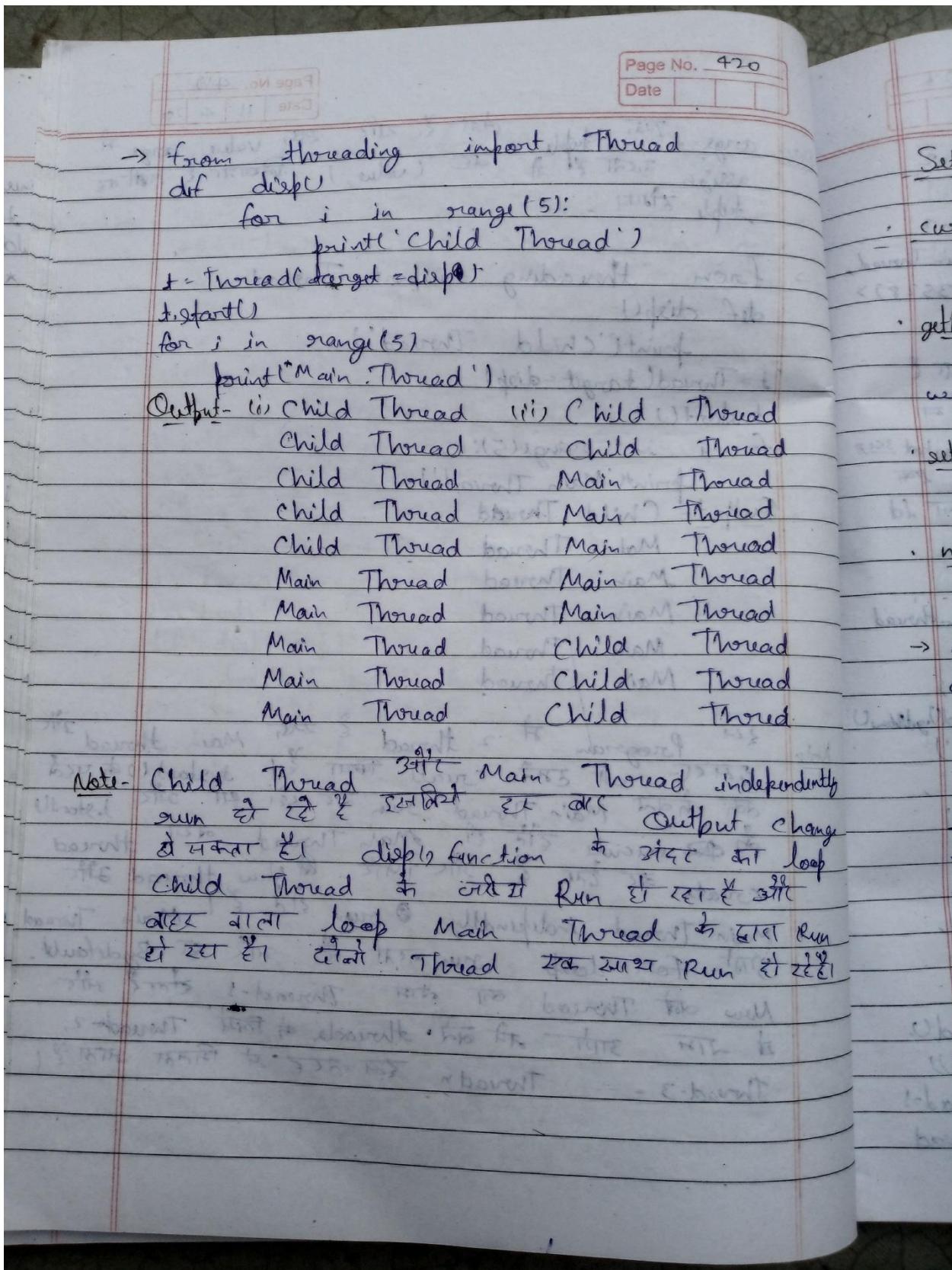
Ex - Tho

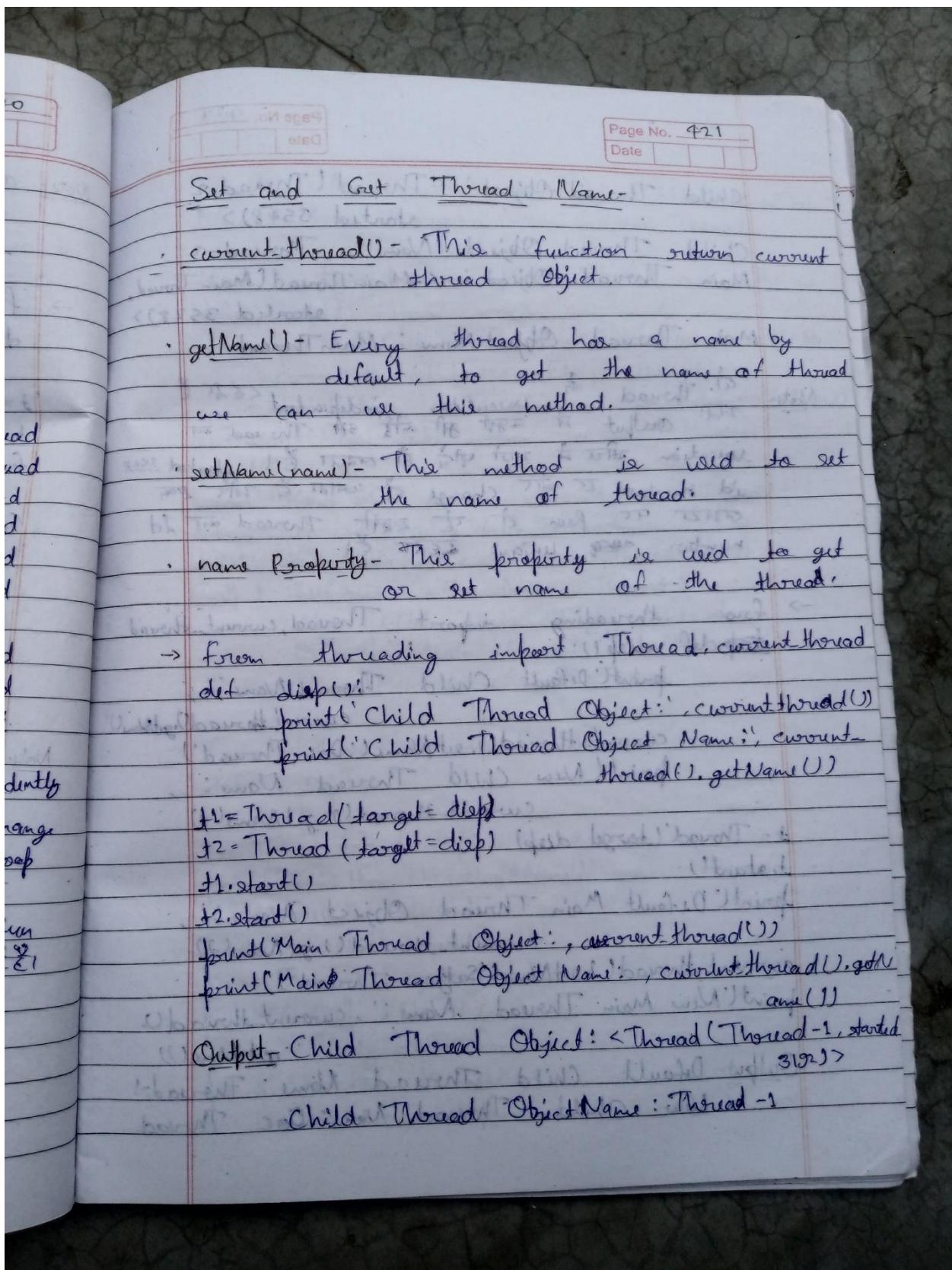
How start

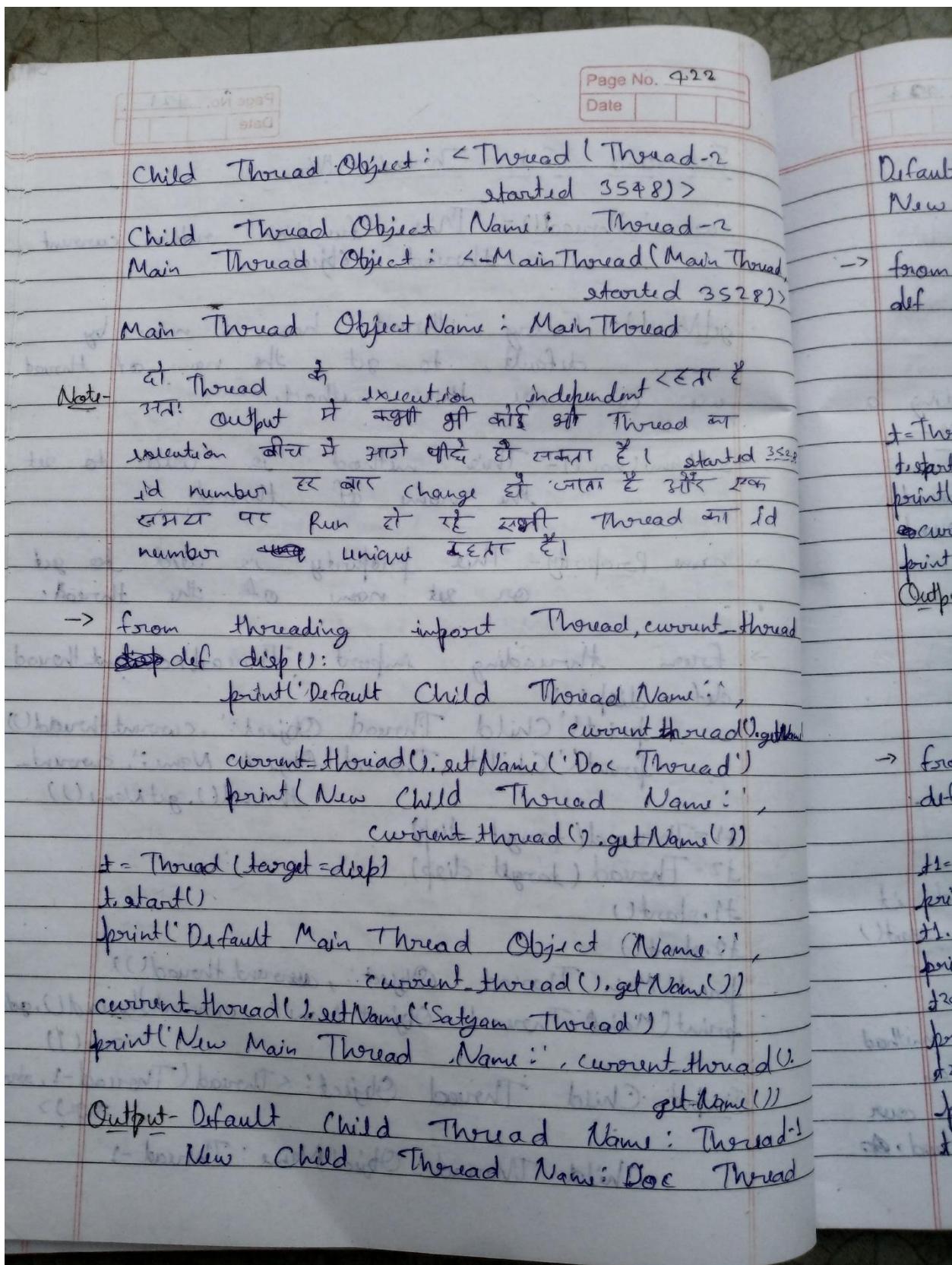


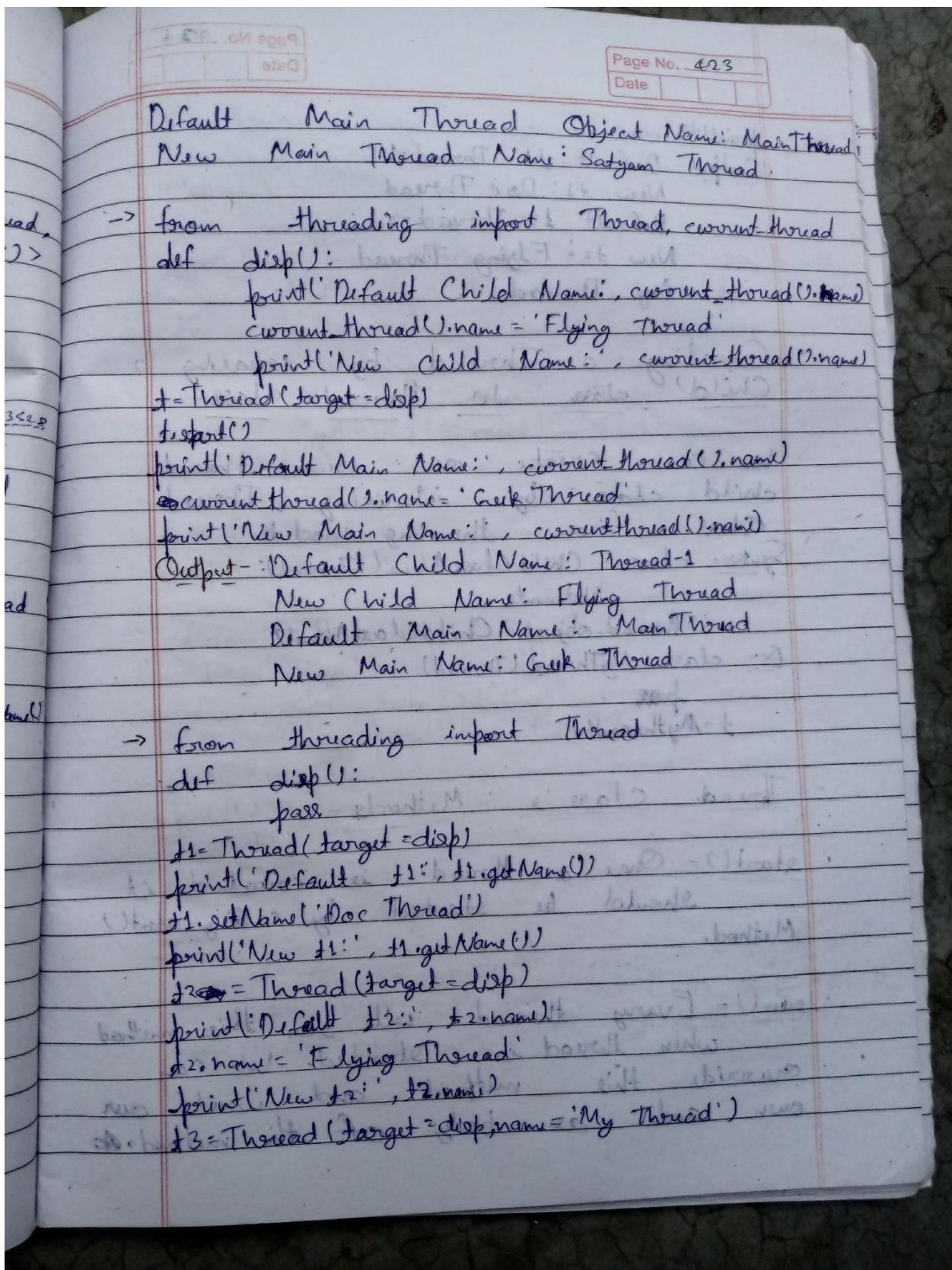












Page No. 424  
Date

`print(t3.name)`

Output - Default : t1: Thread-1  
New t1: Doc Thread  
Default t2: Thread-2  
New t2: Flying Thread  
My Thread

- Creating a Thread by creating a Child class to Thread class -
- We can create our own thread child class by inheriting Thread class from threading module.

Syntax - class ChildClassName(Thread):

statement

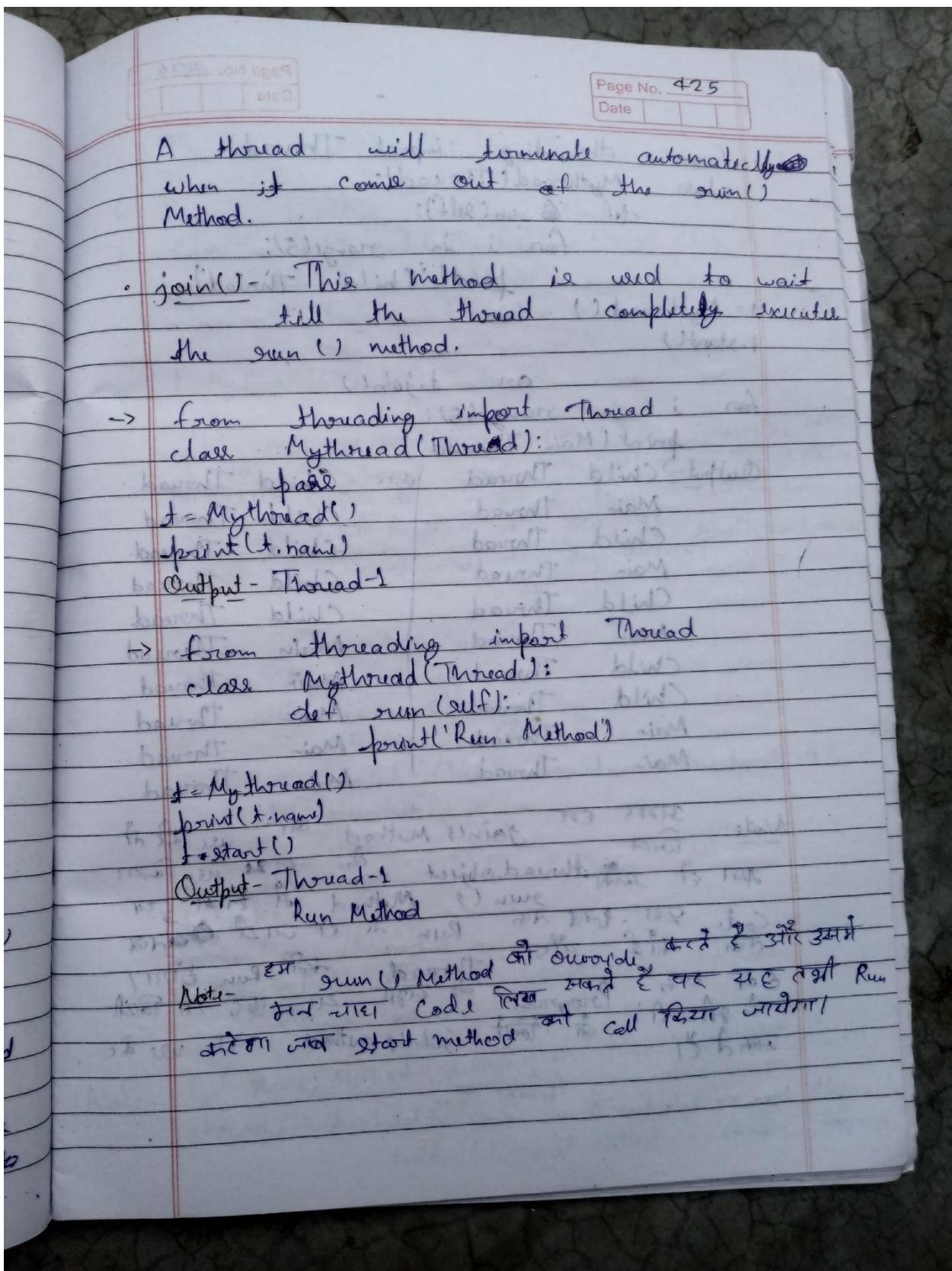
Thread-object = ChildClassName()

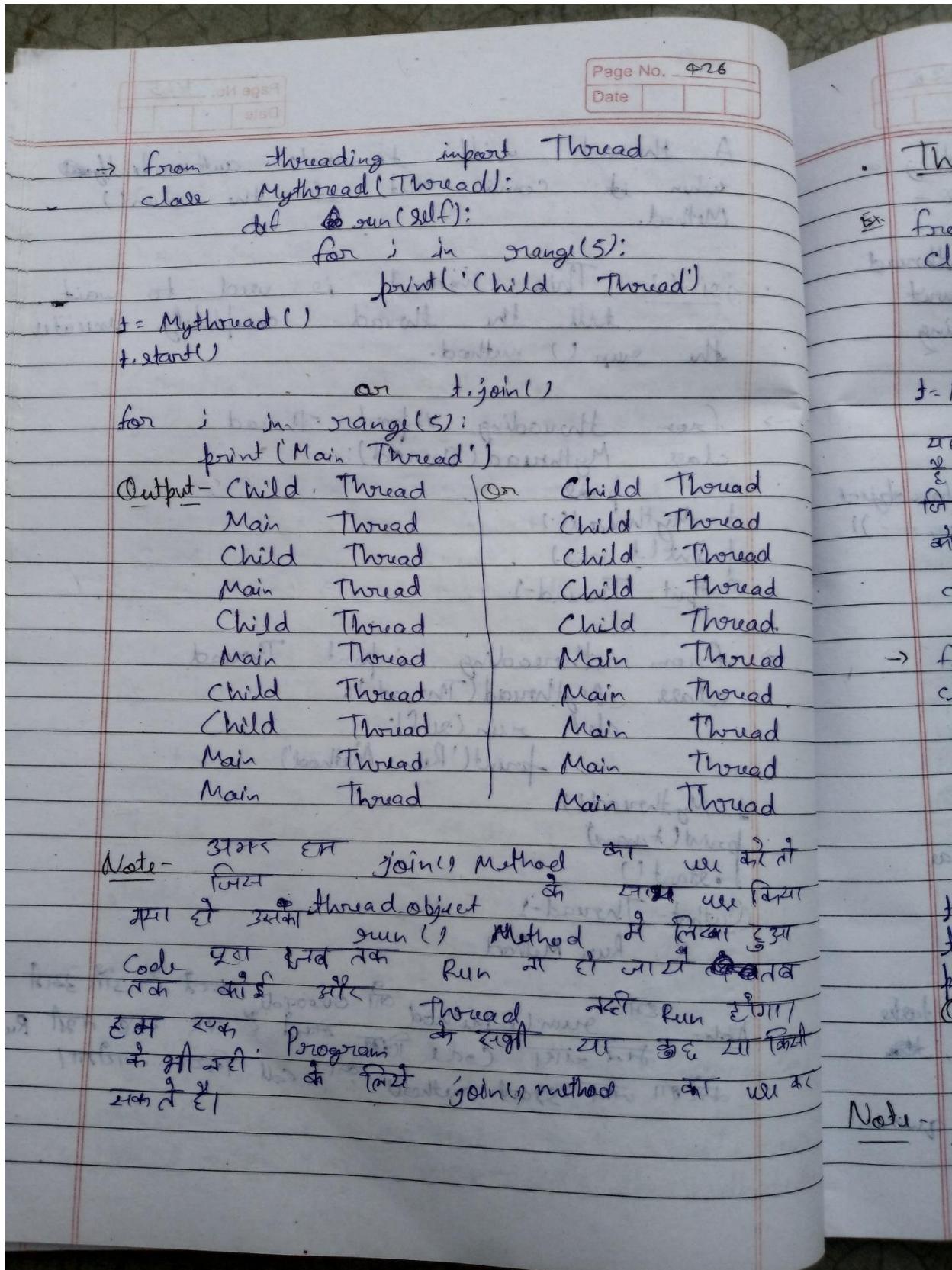
Ex - class Mythread(Thread)

pass

t = Mythread()

- Thread class's Methods -
- start() - Once a thread is created it should be started by calling start() Method.
- run() - Every thread will run this method when thread is started. We can override this method and write our own code as body of the method.





Page No. 427  
Date

Thread Child Class with Constructor.

Ex: from threading import Thread  
 class Mythread(Thread): Thread class as Parent class  
 def \_\_init\_\_(self, a):  
 Thread.\_\_init\_\_(self) Calling Thread class constructor  
 self.a=a

t = Mythread(10)

Mythread class, Thread class inheritance  
 Mythread class constructor at line 2  
 for thread & for all Thread class & constructor  
 call start at line 2 at line 4 Mythread class  
 class constructor at line 4

→ from threading import Thread  
 class Mythread(Thread):  
 def \_\_init\_\_(self, a):  
 Thread.\_\_init\_\_(self)  
 print('Child Thread Constructor', a)  
 def run(self):  
 pass

t = Mythread(10)  
 t.start()  
 print('Main Thread')

Output- Child Thread Constructor 10  
Main Thread.

Note: - ~~run() Method define acoording to thread~~  
 Thread.\_\_init\_\_(self) at line 2 Runtime error: threads  
 \_\_init\_\_() not called , error ~~at line 4~~

Page No. 428  
Date

- Creating a thread without creating a child class to Thread class -

We can create an independent thread child class that does not inherit from Thread class from threading module.

Syntax - class ClassName:

```

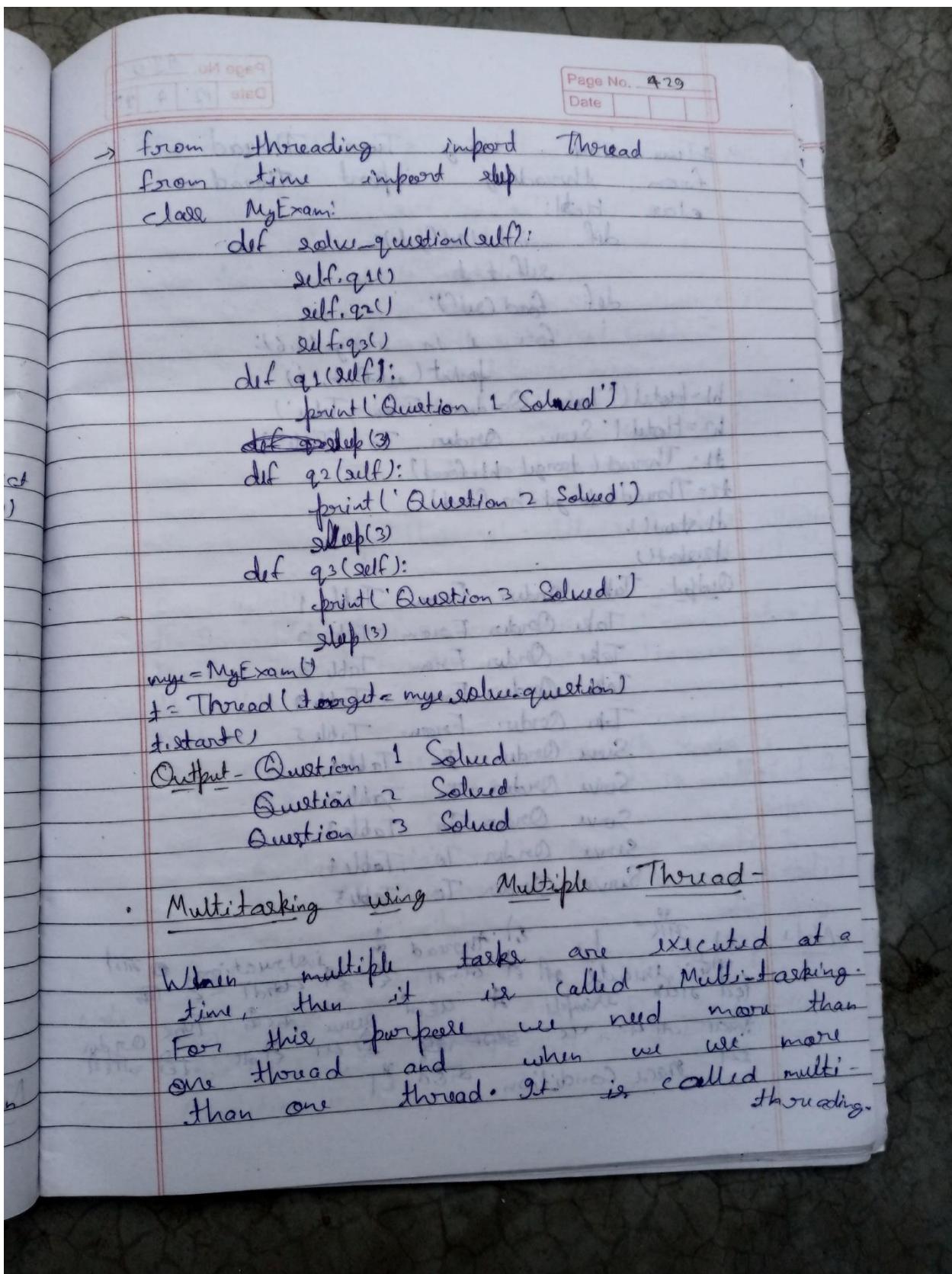
class ObjectName = ClassName()
    statements
    ThreadObject = Thread(target = classObject
                           .name.functionName, args = (arg1, arg2, ...))
    
```

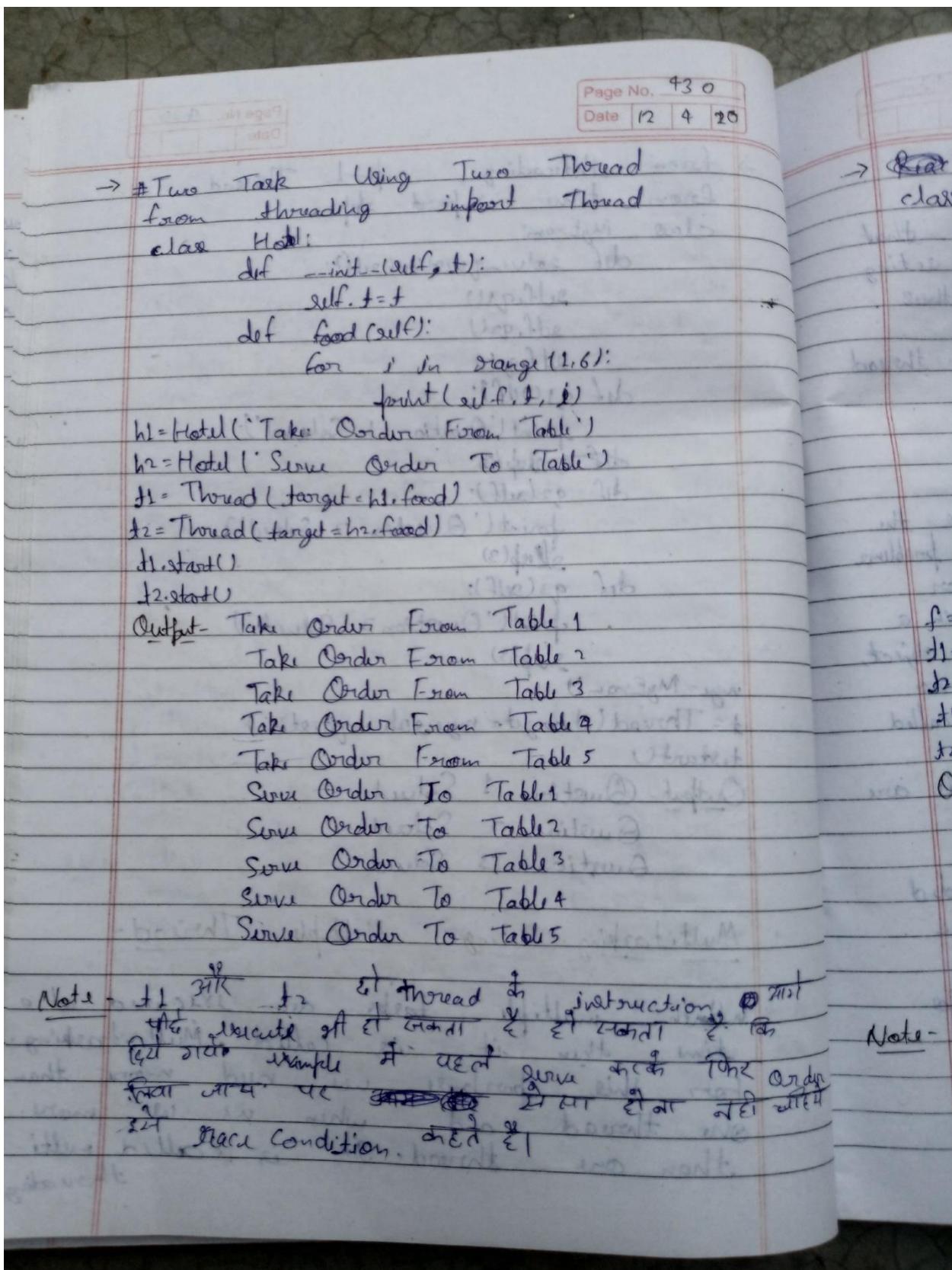
→ from threading import Thread  
 class MyThread:  
 def disp(self, a, b):  
 myt = MyThread()  
 t = Thread(target = myt.disp, args = (10, 20))  
 t.start()

(Output - 10 20)

Note: disp in first self is for object name as argument args of static method & t.

- Single Tasking using a Thread - When multiple tasks are executed by a thread one by one, then it is called single tasking.  
Ex - writing examination we solve all question one by one Ques1, Ques2, Ques3, ...





```

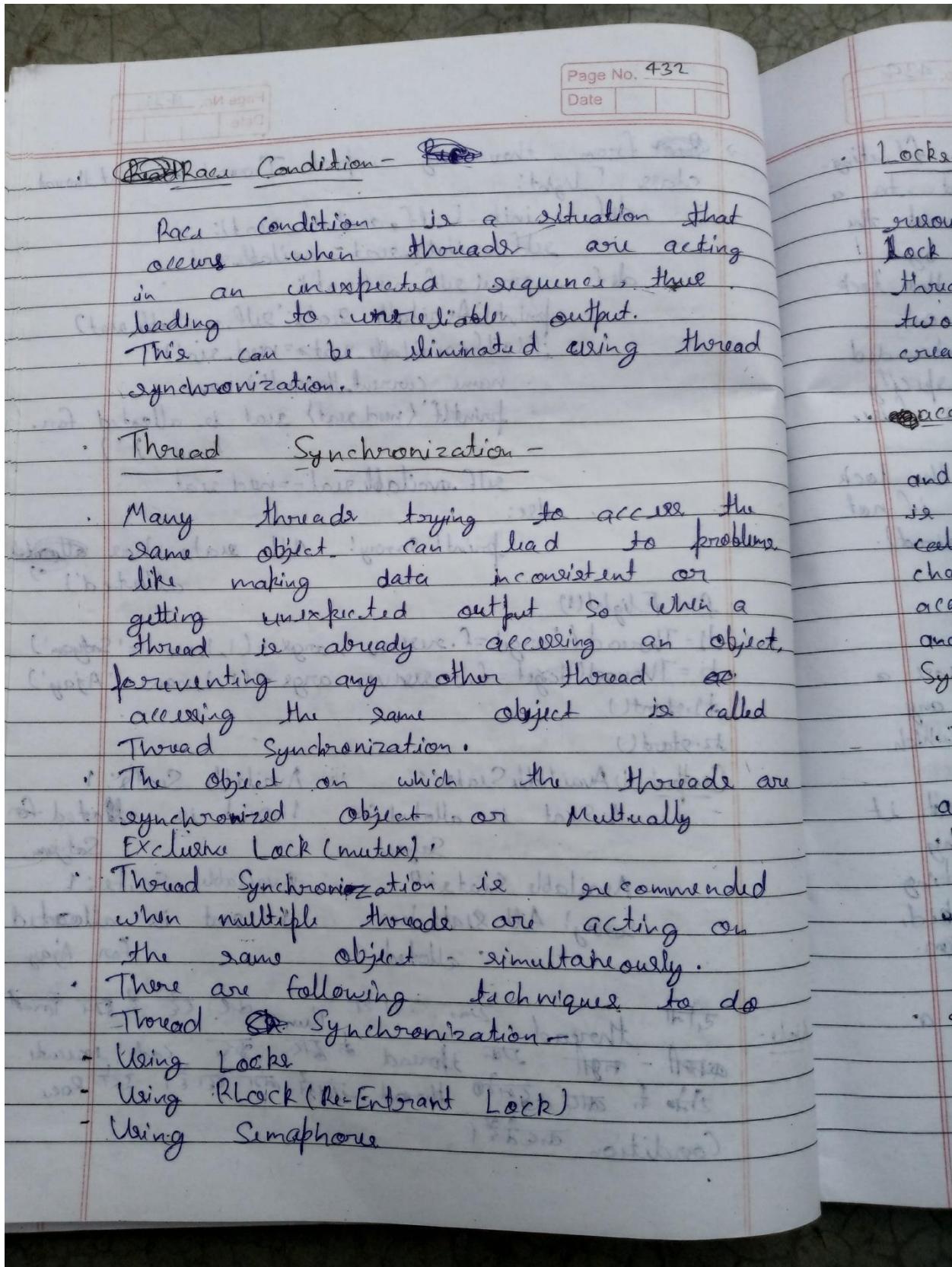
→ from from threading import Thread, current_thread
class Flight:
    def __init__(self, available_seat):
        self.available_seat = available_seat
    def new_seat(self, need_seat):
        print('Available Seats:', self.available_seat)
        if self.available_seat >= need_seat:
            name = current_thread().name
            print(f'{need_seat} seat is allocated for {name}')
            self.available_seat -= need_seat
        else:
            print('Sorry! All seats have been allocated')

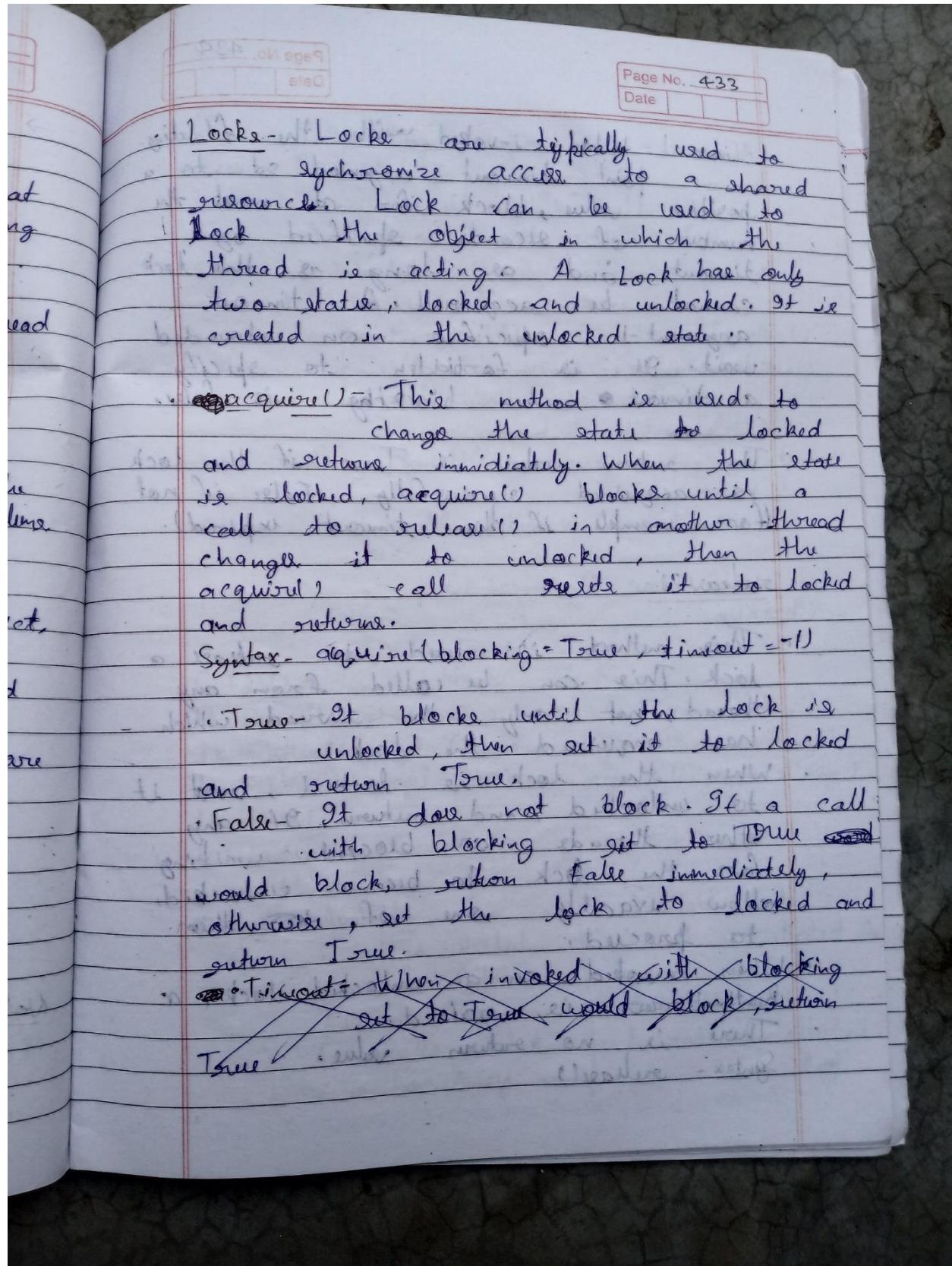
```

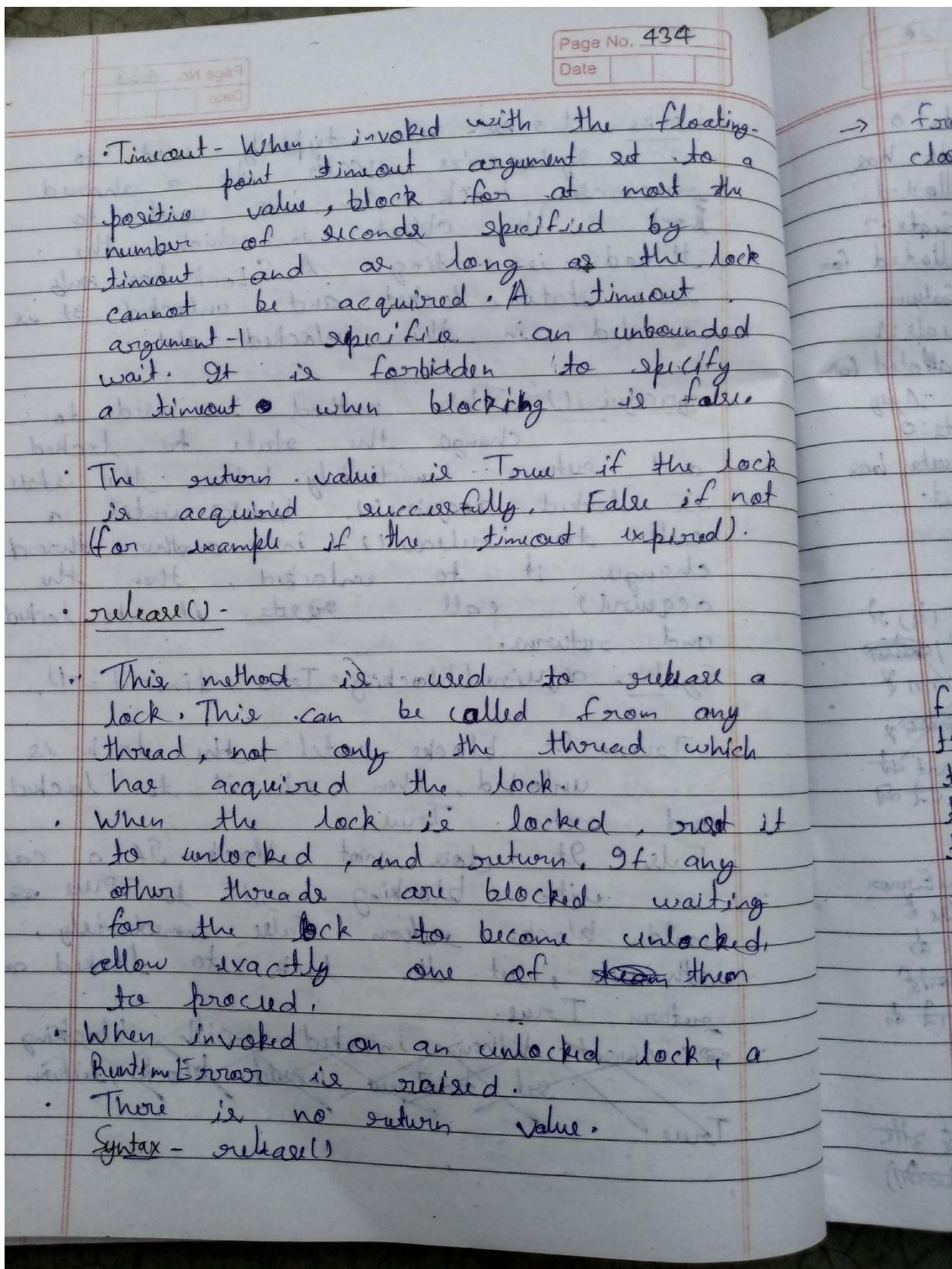
```
f = Elight()  
t1 = Thread(target=f, name='Sathyam', args=(1,))  
t2 = Thread(target=f, name='Ajay', args=(1,))  
t1.start()  
t2.start()
```

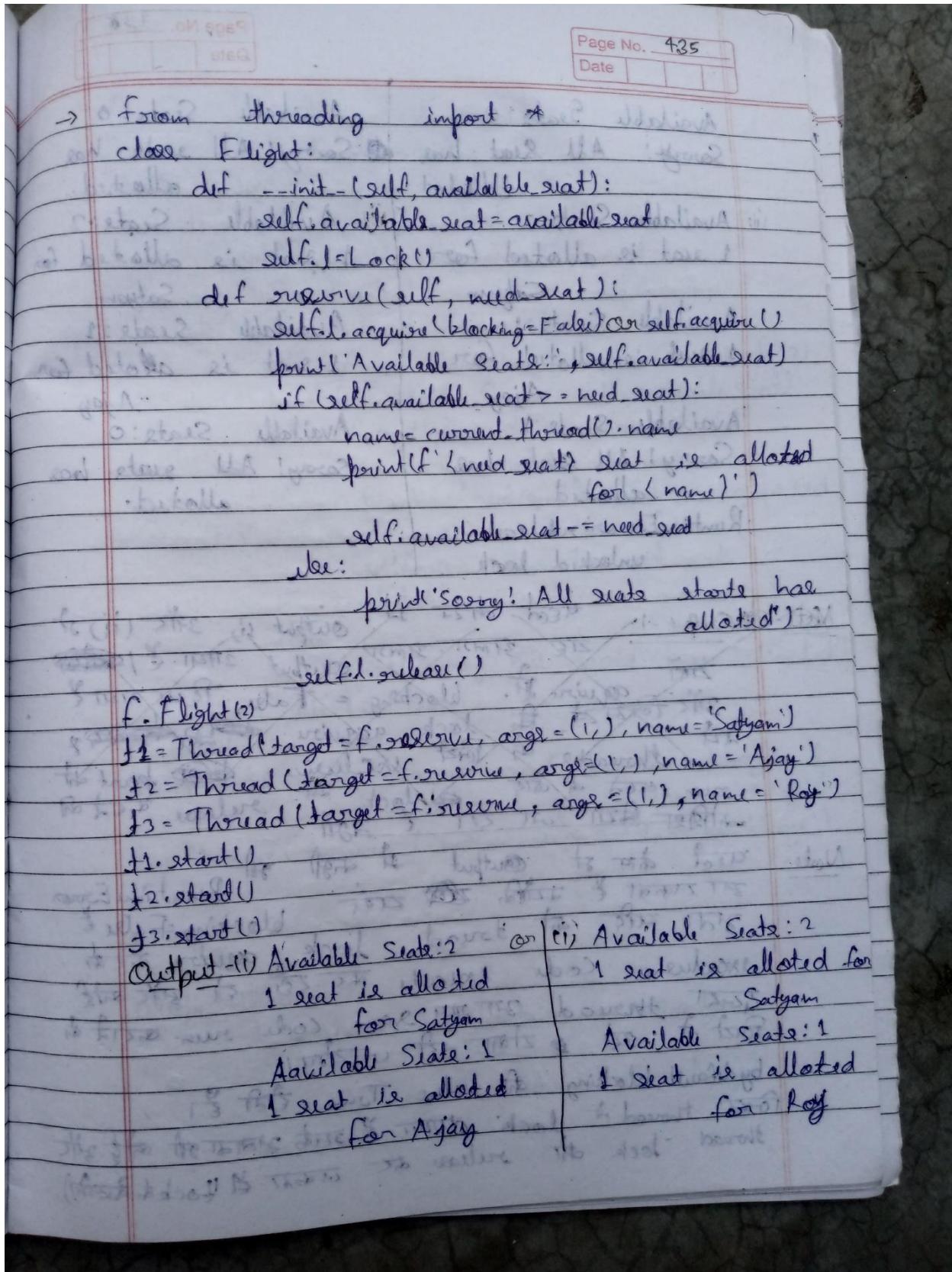
Output - ii) Available Seats: 1      (ii) Available Seats: 1  
1 seat is allotted for 1 seat is allotted for  
Satyam Satyam  
Available Seats: 0 Available Seats: 1  
Sorry! All seats have 1 seat is allotted  
allocated for Ajay

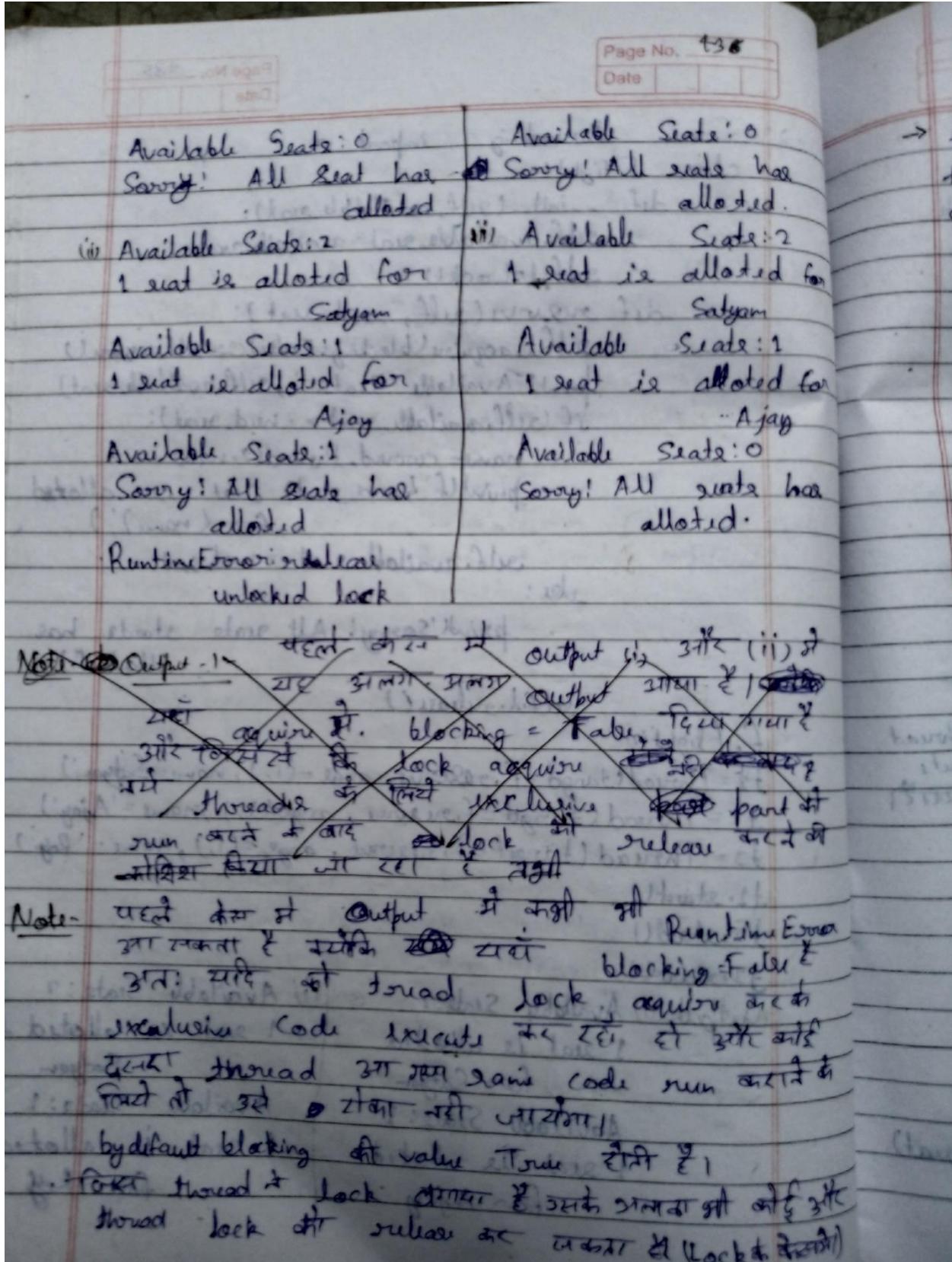
Note - दौनी - thorax Run the 22 ₹ 521 Total  
अंगी - wing Its thorax के पास छोटे code execute  
जीम के बाद उसके Thorax पर लग होता है 1st Race  
Condition में होता है।











Page No. 427  
Date \_\_\_\_\_

```

→ from threading import *
from time import sleep
class Flight:
    def __init__(self, available_seat):
        self.available_seat = available_seat
        self.l = Lock()
    def reserve(self, need_seat):
        self.l.acquire(blocking=True, timeout=2)
        print("Available Seats: " + str(self.available_seat))
        if (self.available_seat >= need_seat):
            name = current_thread().name
            print(f"({need_seat}) seat is allotted for {name}")
            self.available_seat -= need_seat
            sleep(4)
        else:
            print("Sorry! All seats have allotted")
        self.l.release()
    f = Flight(2)
    t1 = Thread(target=f.reserve, args=(1,), name='Satyam')
    t2 = Thread(target=f.reserve, args=(1,), name='Ajay')
    t3 = Thread(target=f.reserve, args=(1,), name='Roy')
    t1.start()
    t2.start()
    t3.start()

```

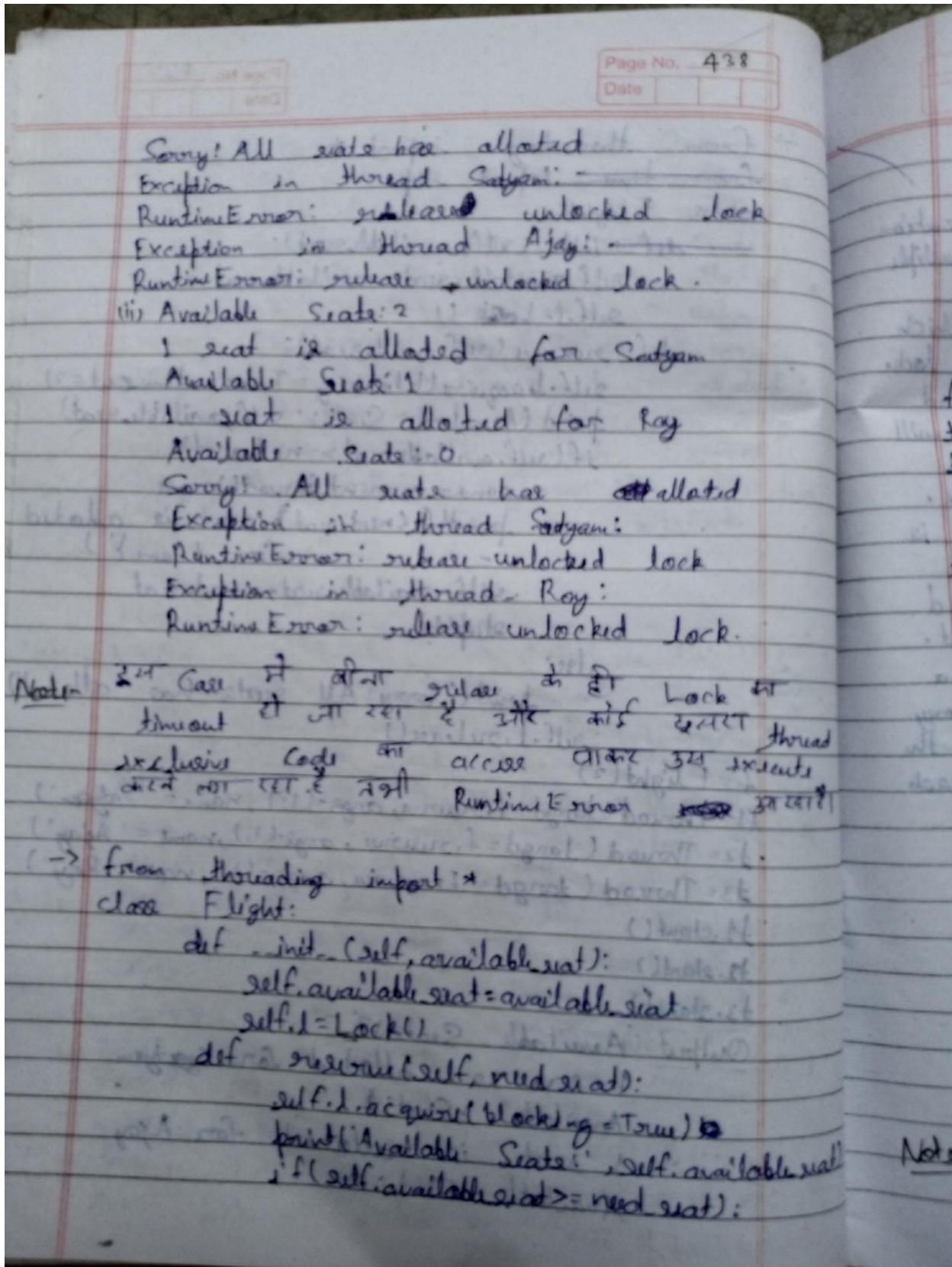
Output - i) Available Seats:

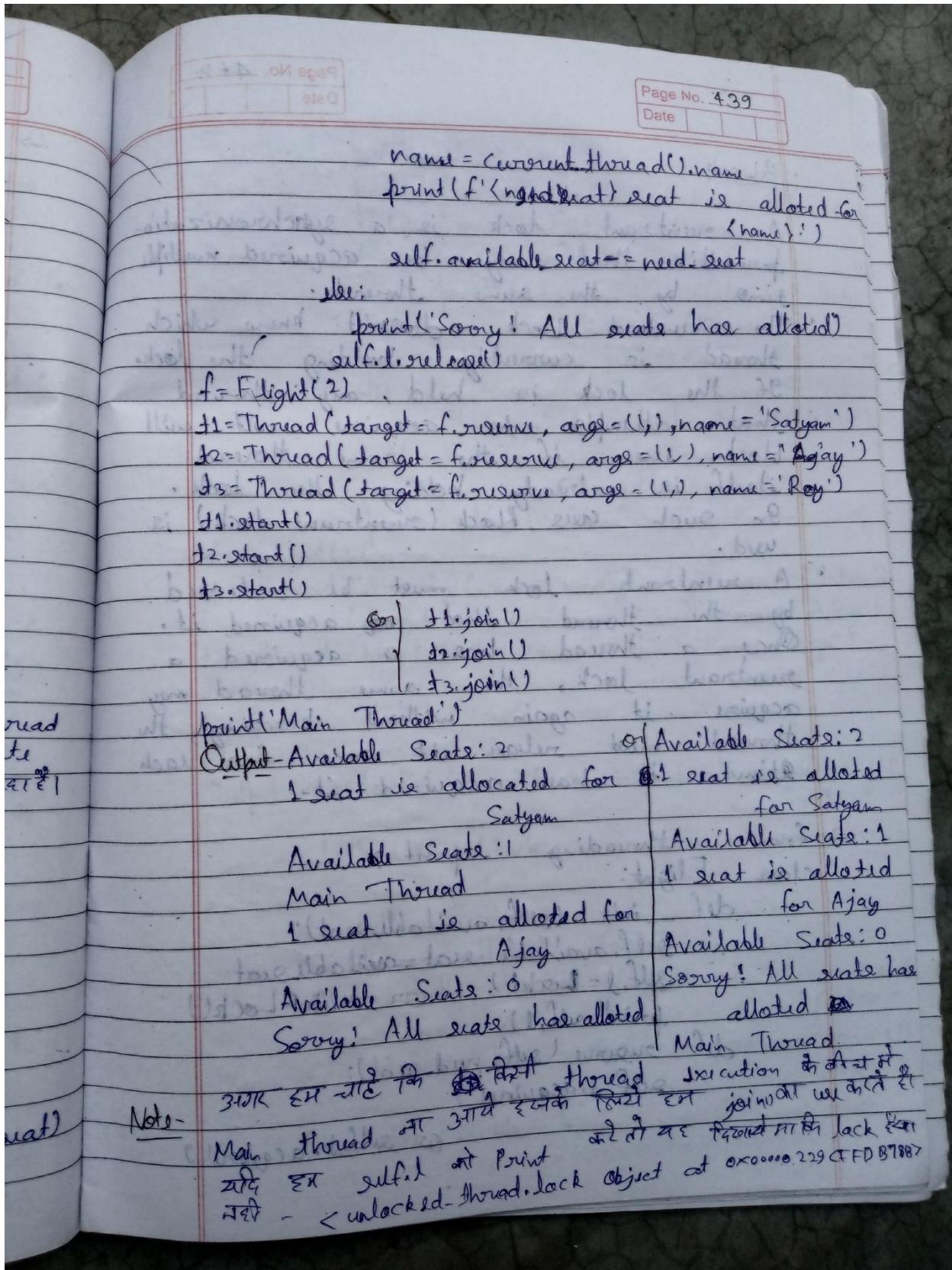
1 seat is allotted for Satyam

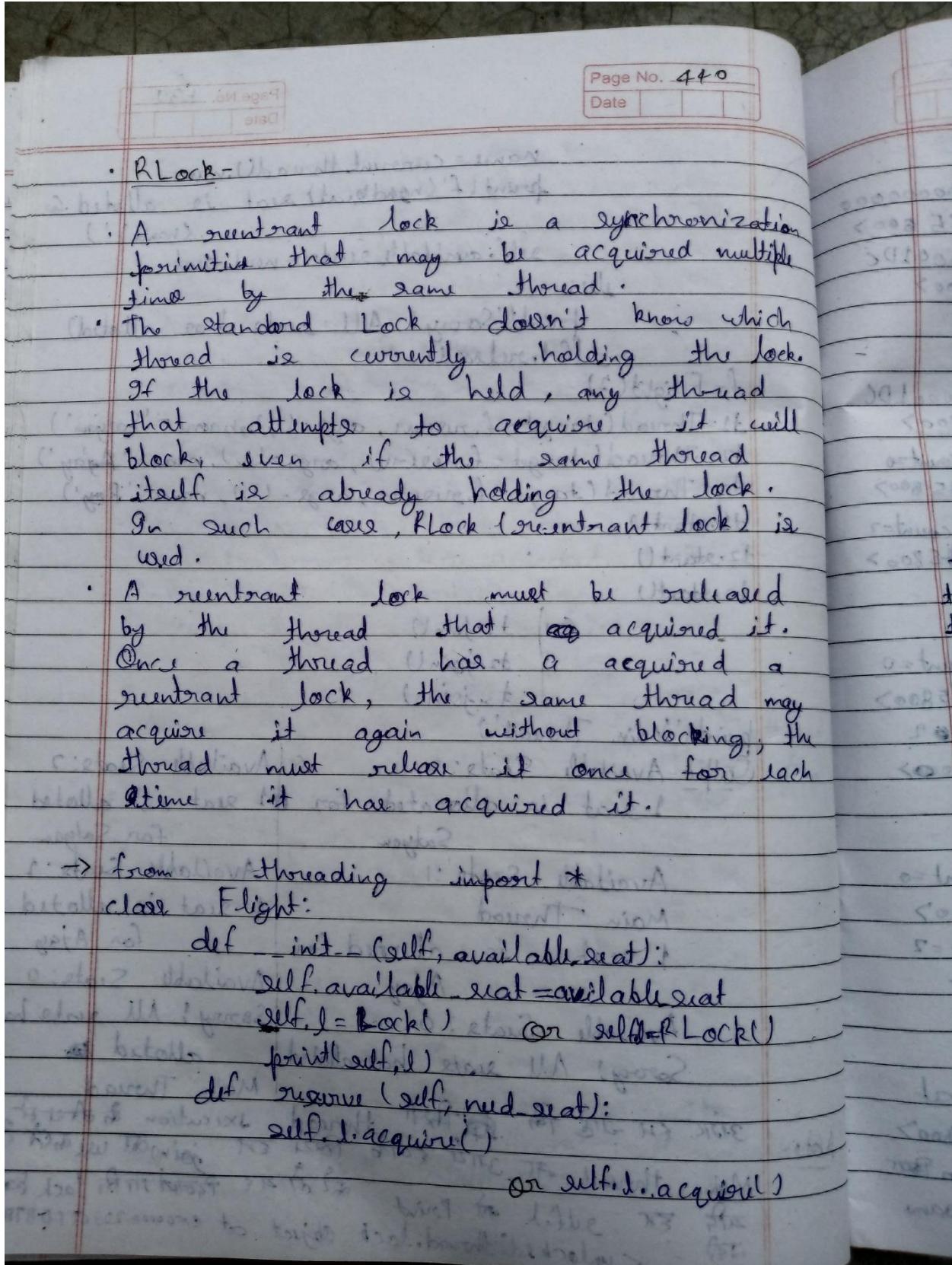
Available Seats:

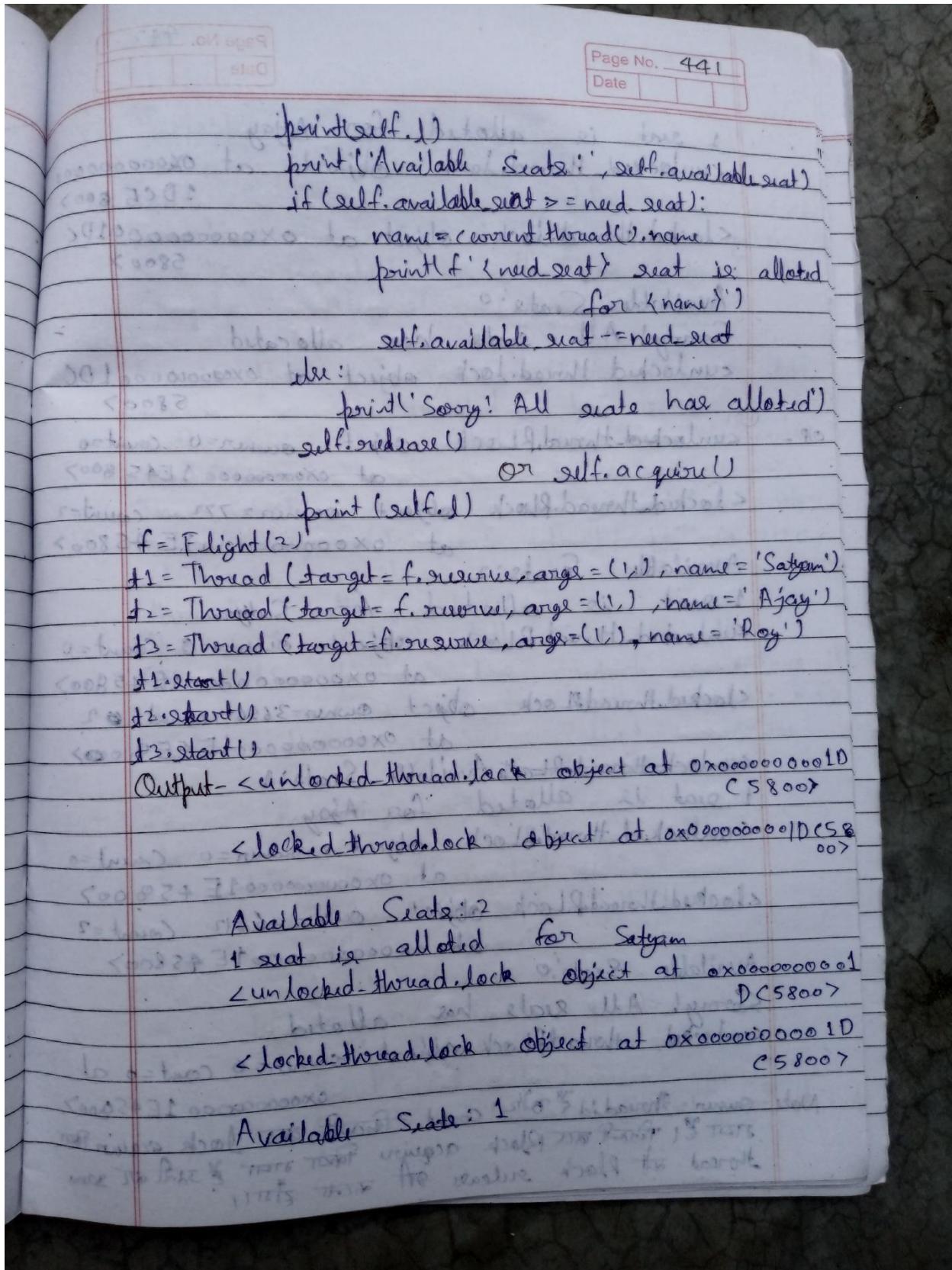
1 seat is allotted for Ajay

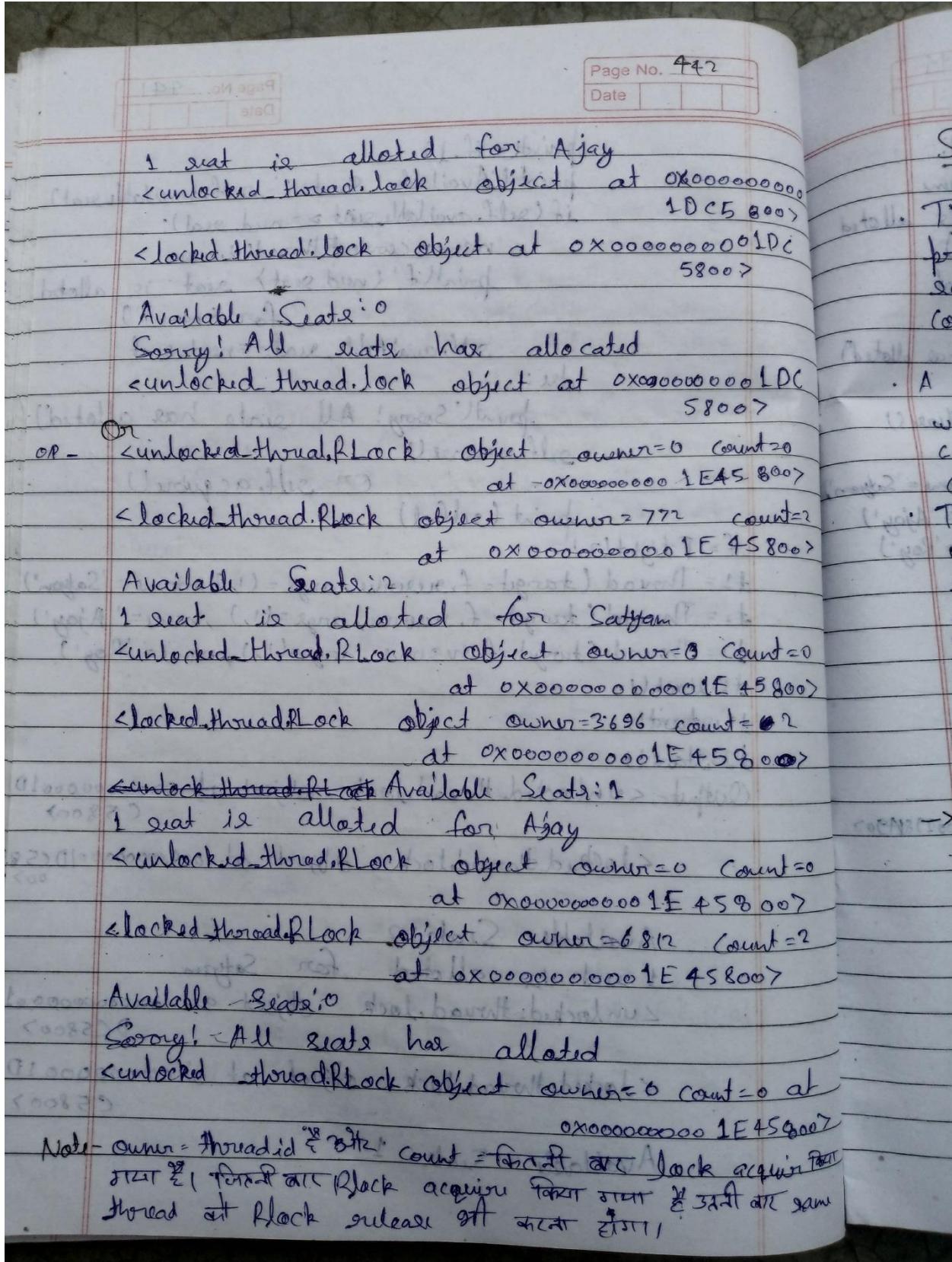
Available Seats: 0

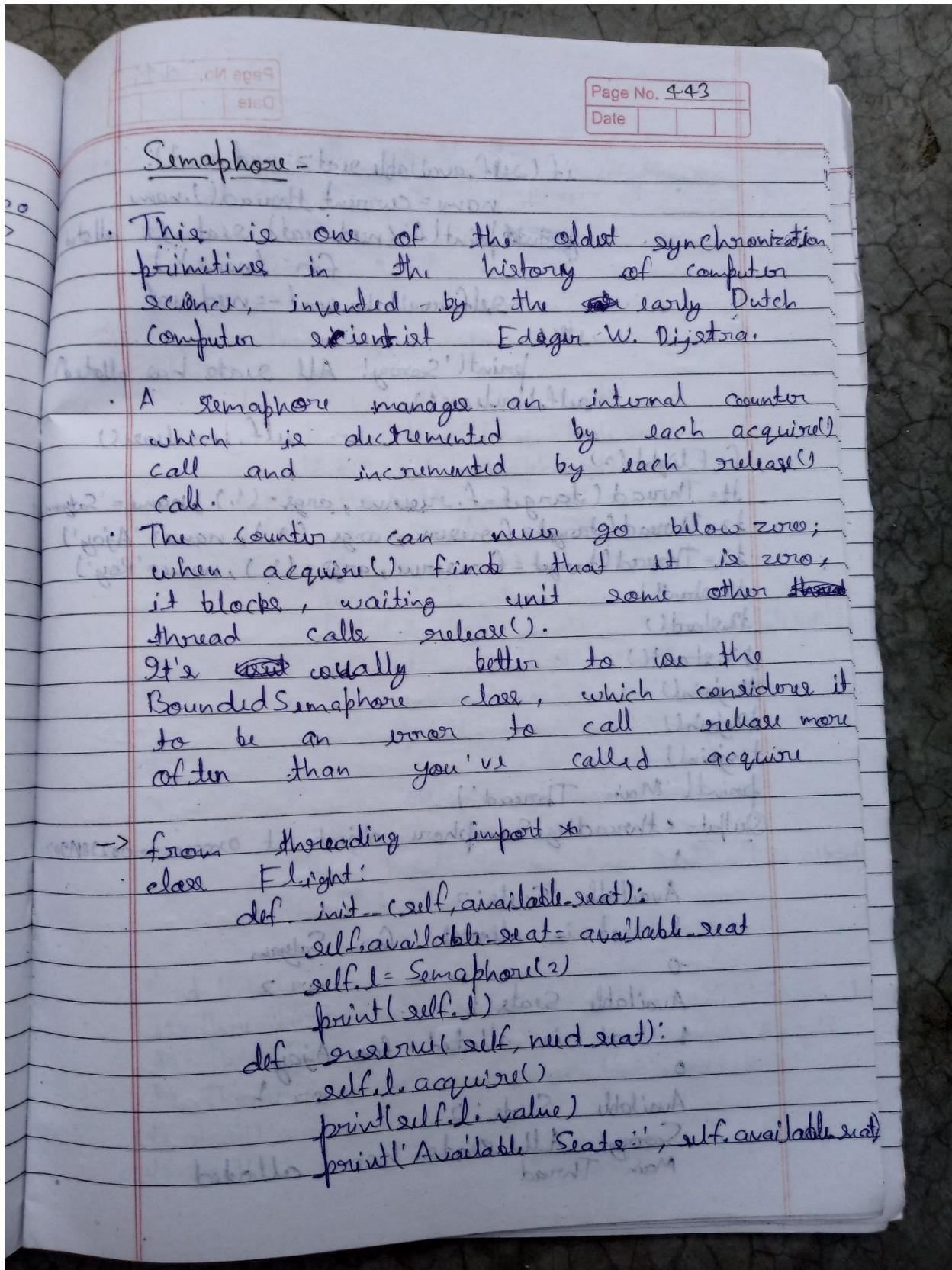


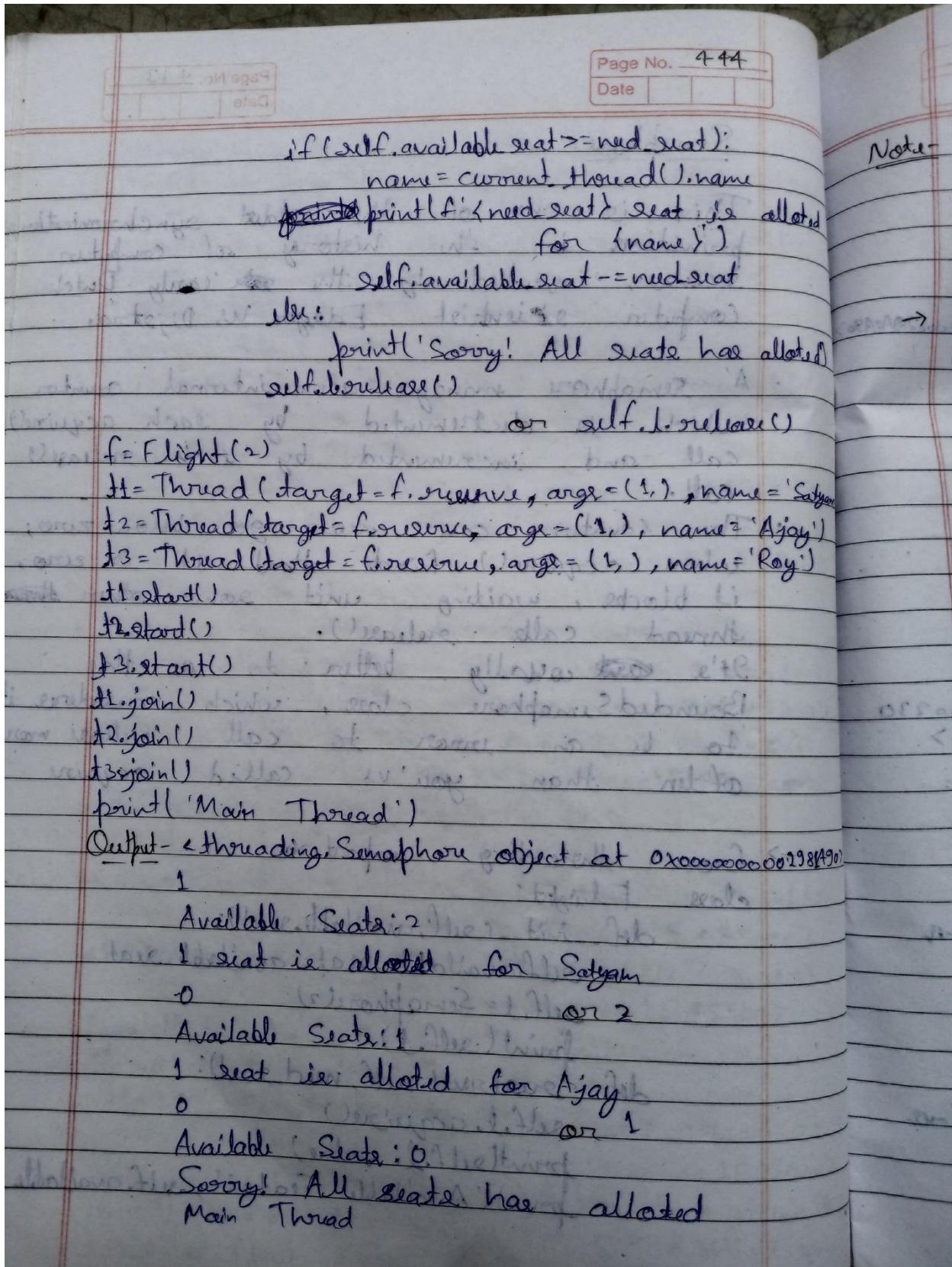


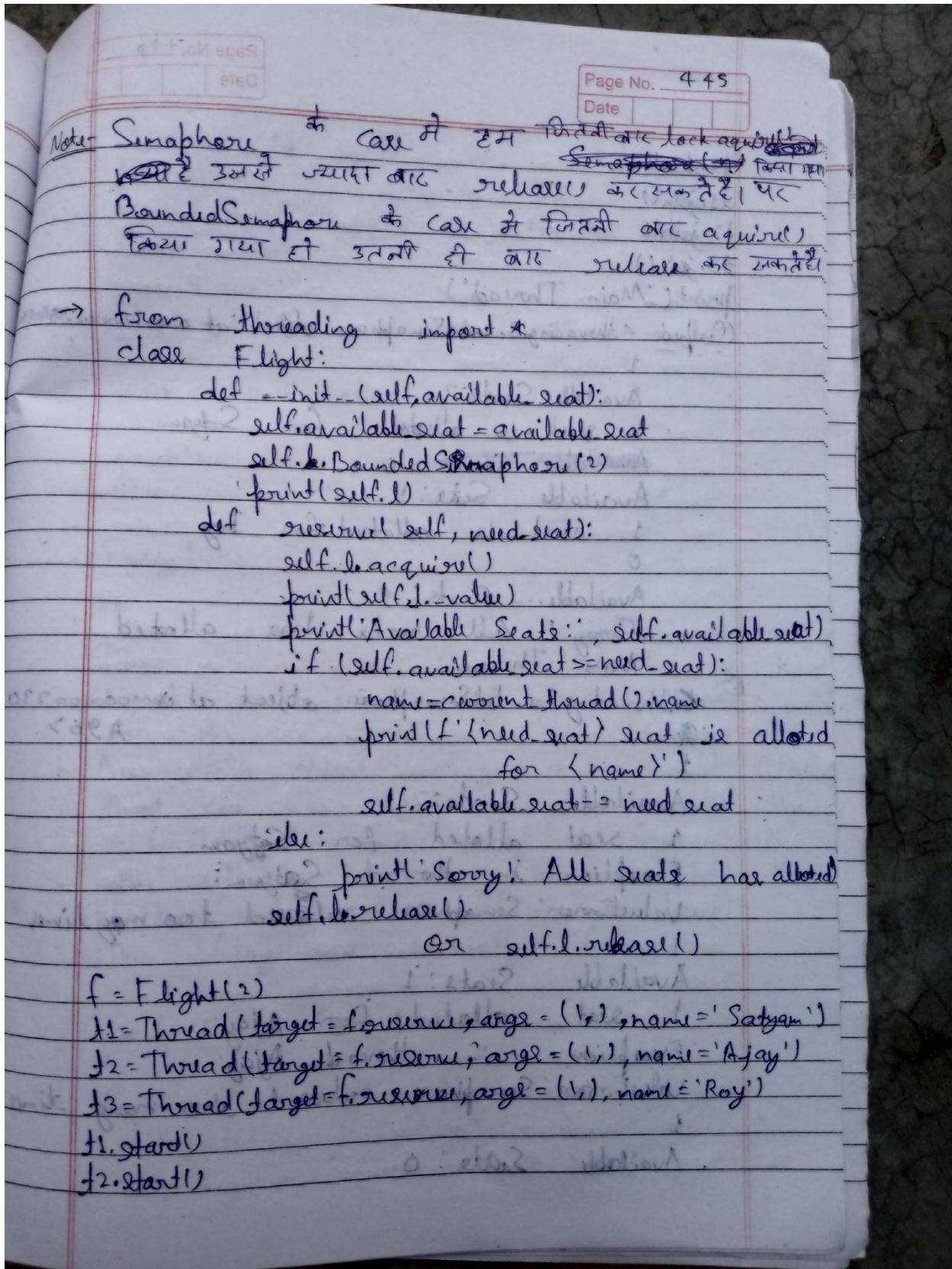


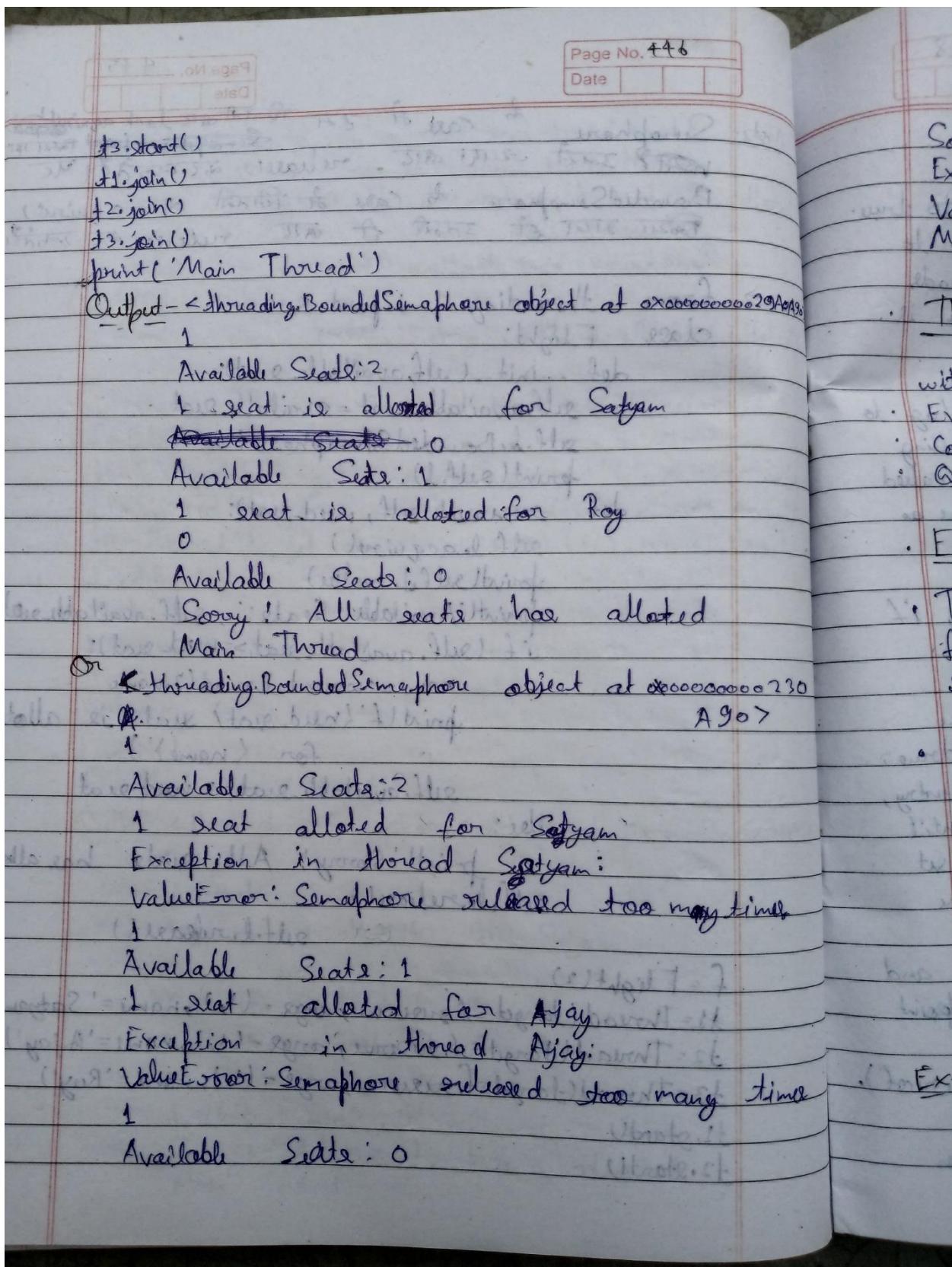


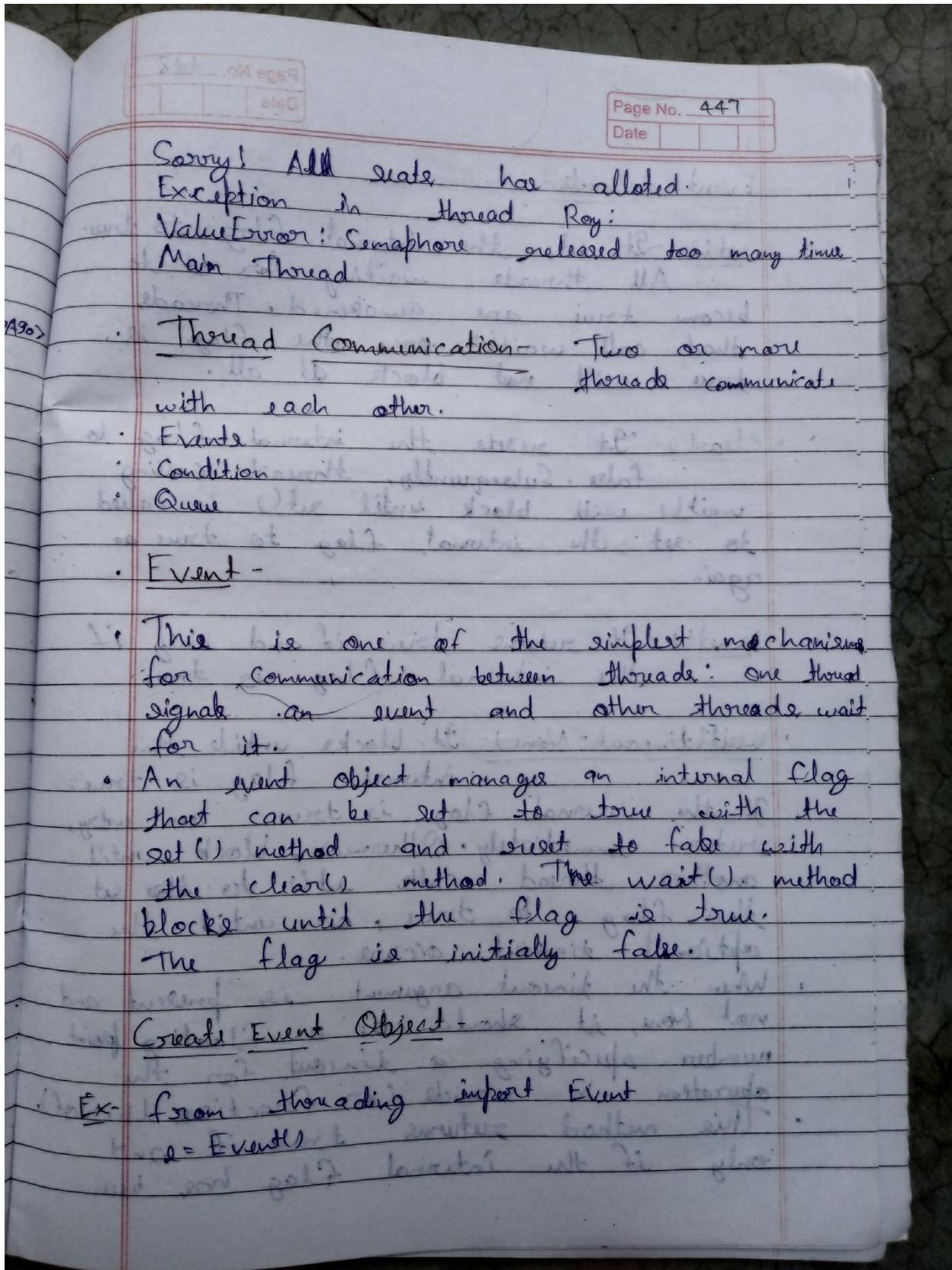


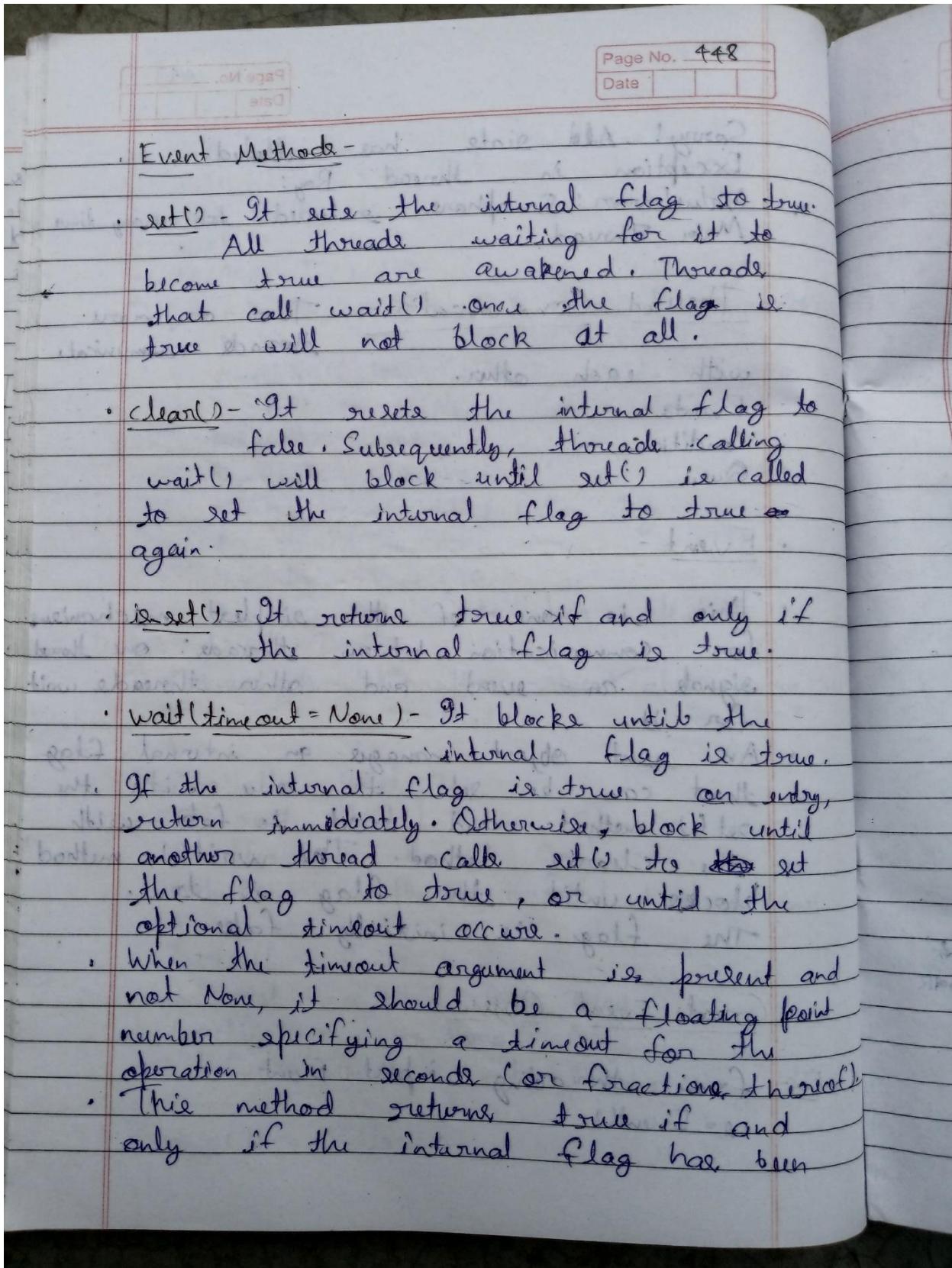


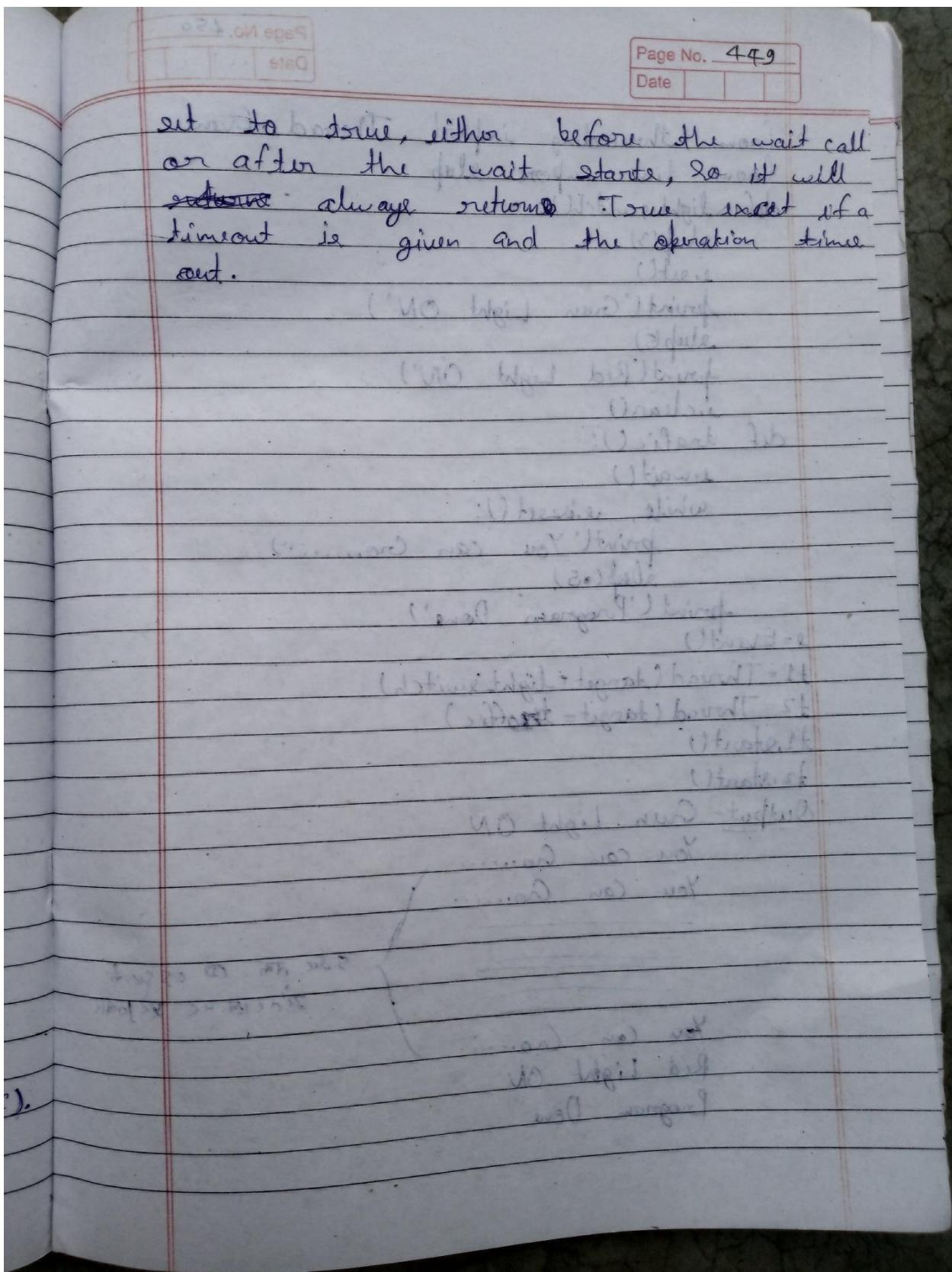


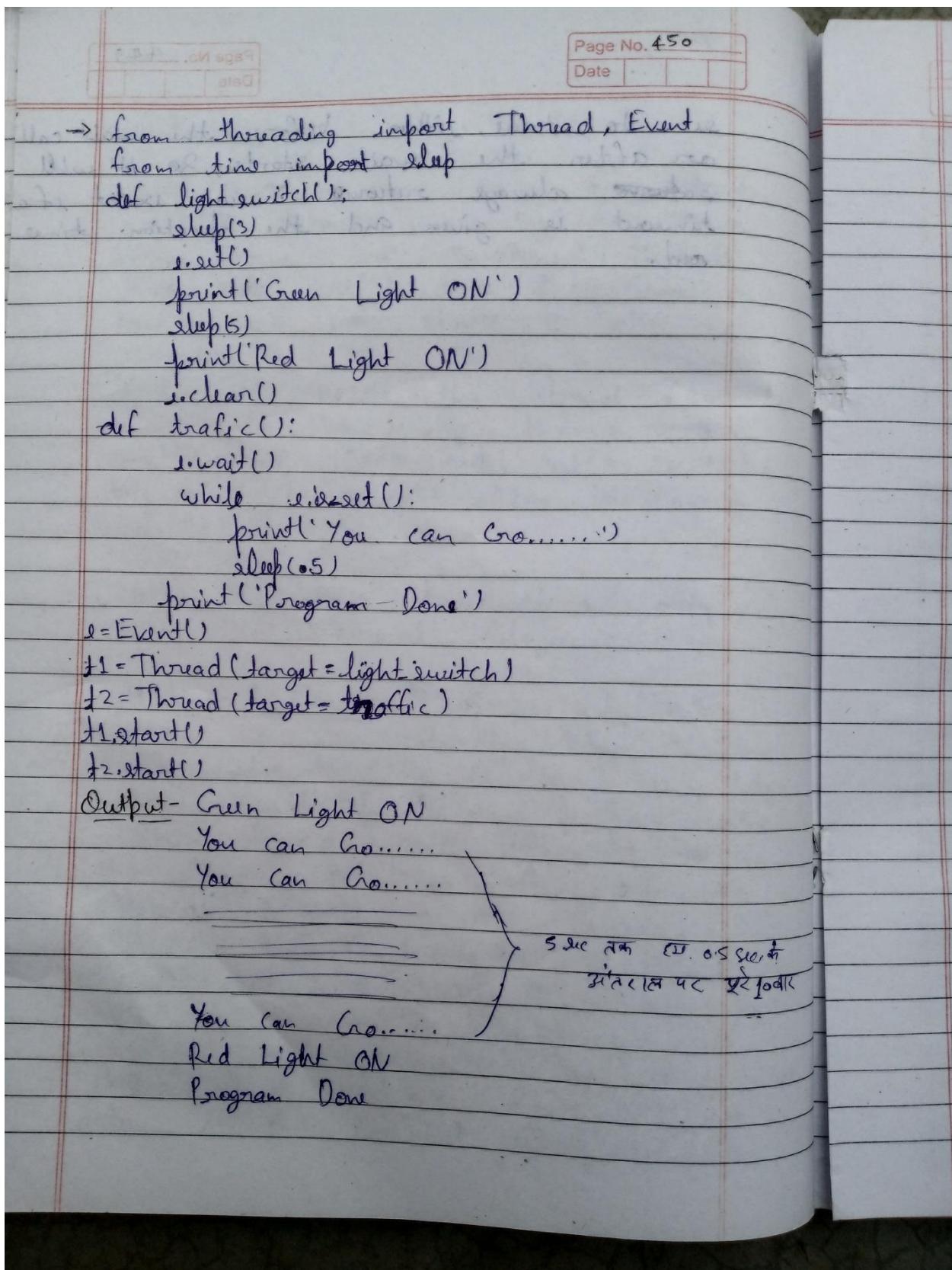








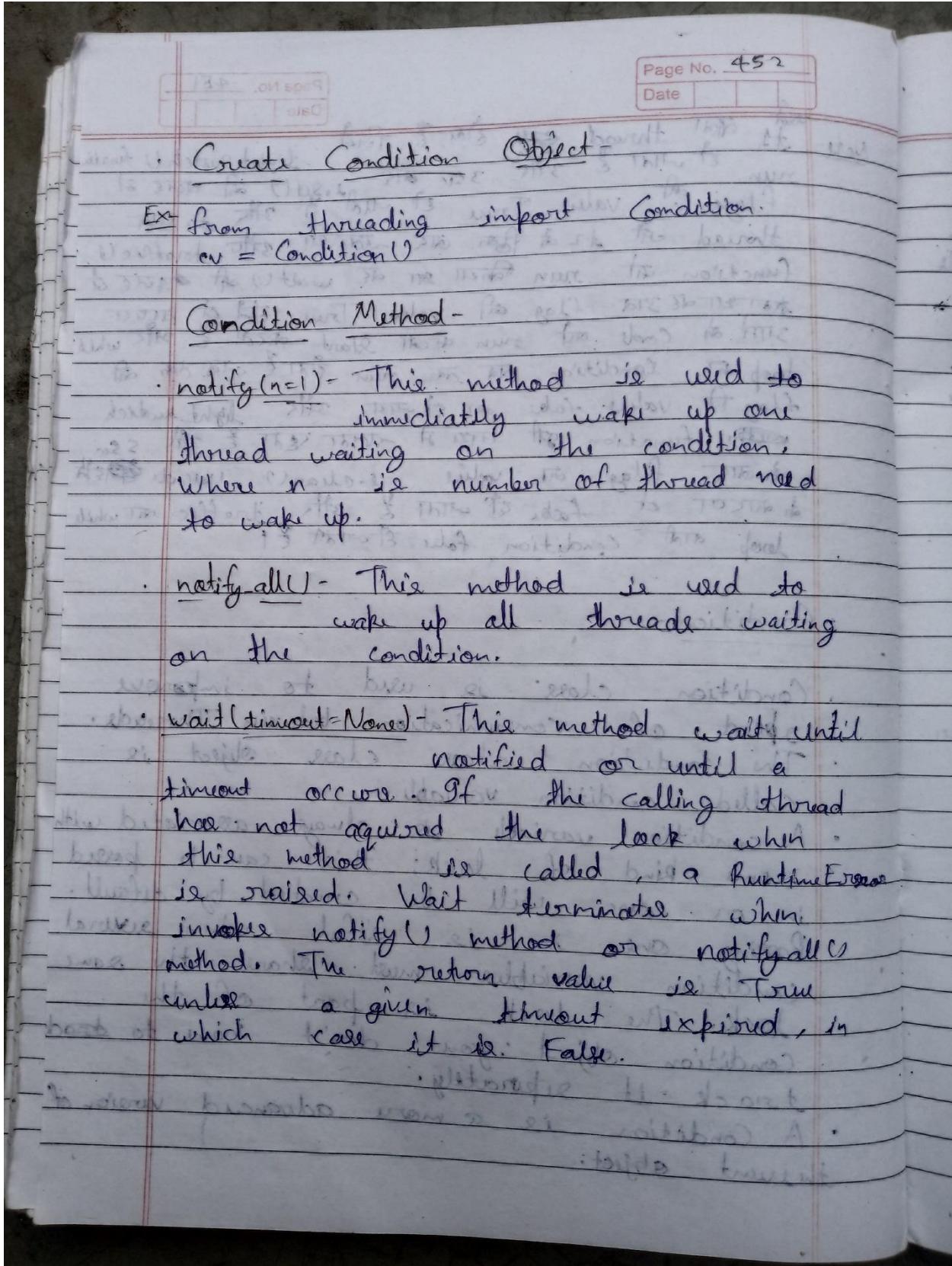


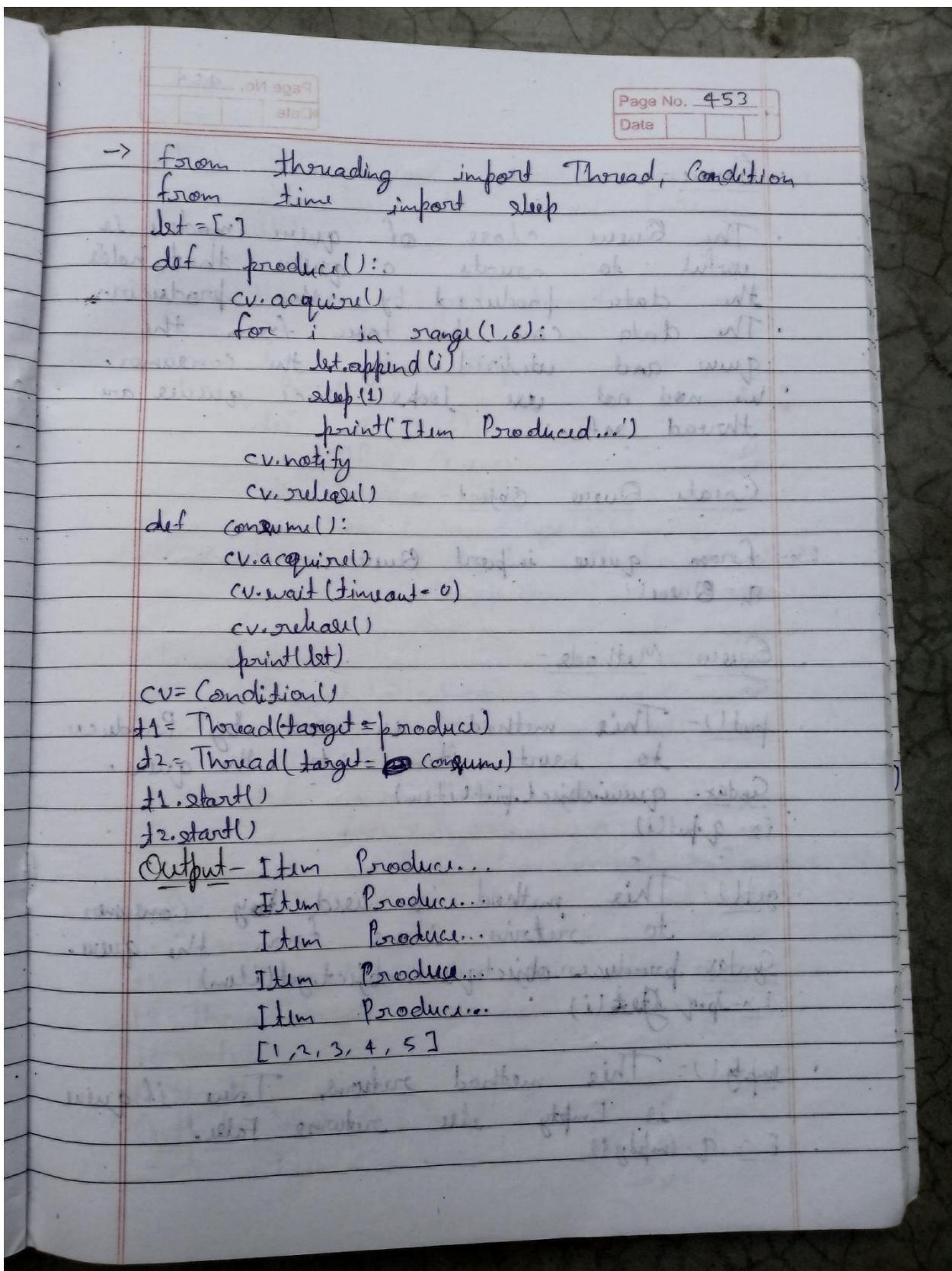


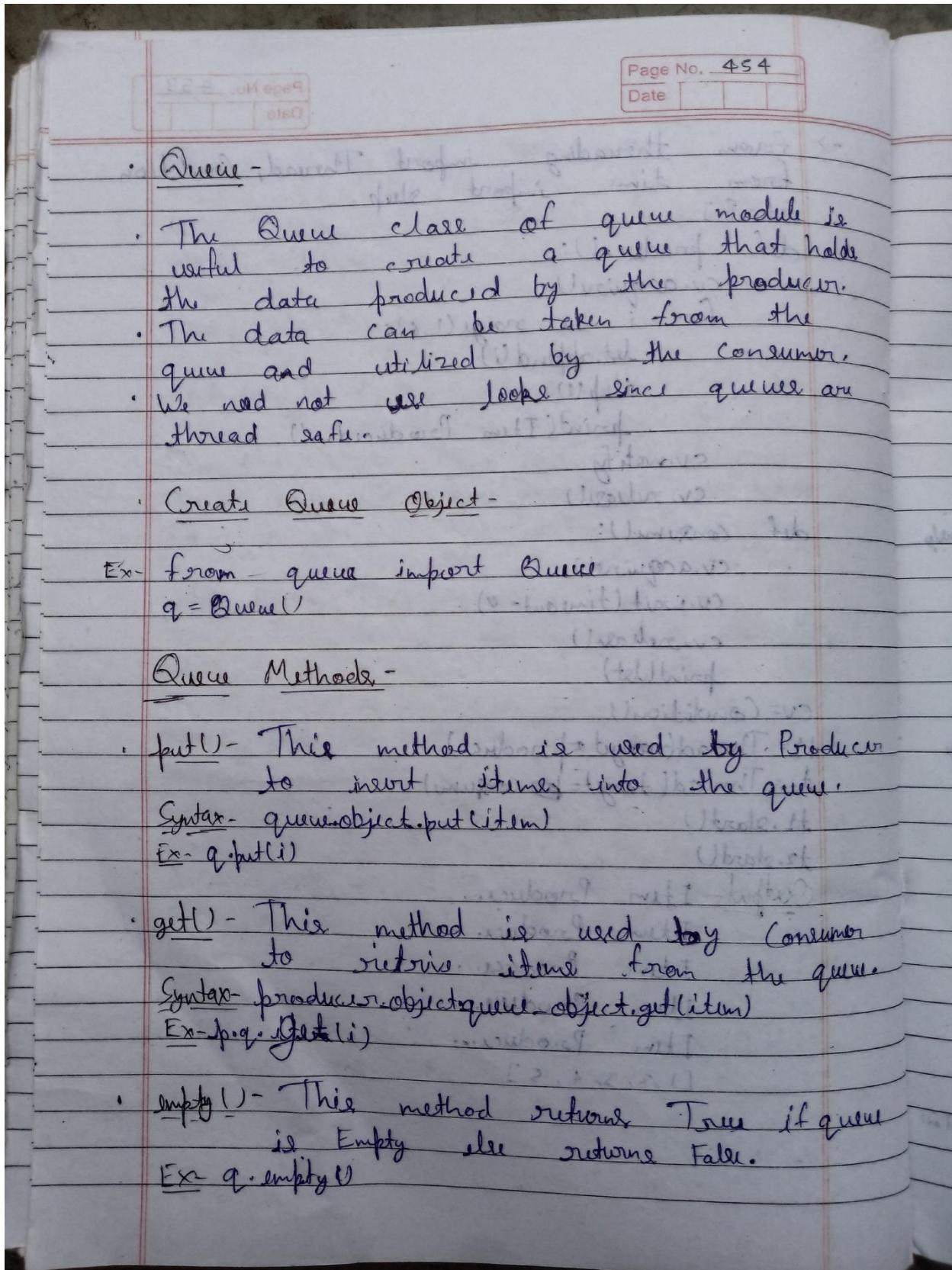
Note- #1 वेला thread की होती है पर्ति light-switch U function  
run की जाती है 3rd 3sec बाद 3sec की वजह से  
flag की value True हो जाती है 3rd & 4th  
thread की 4th की तिक बाद -> तो यह 3rd traffic()  
function का run fuction भी है wait() की वजह से  
क्रमान्वयन आगे flag की value True होती है बाहर  
आए के Code का run करते start करते हैं और while  
loop की condition तब तक run होती है तब तक की  
flag की value false ना हो पाये और light switch  
function की जाये तो अलाइट होता है और 5 sec.  
की बाद flag की value 1-clear() इन्टेंसिटी  
की कारण से false हो जाता है और traffic का while  
loop तभी condition false हो जाता है।

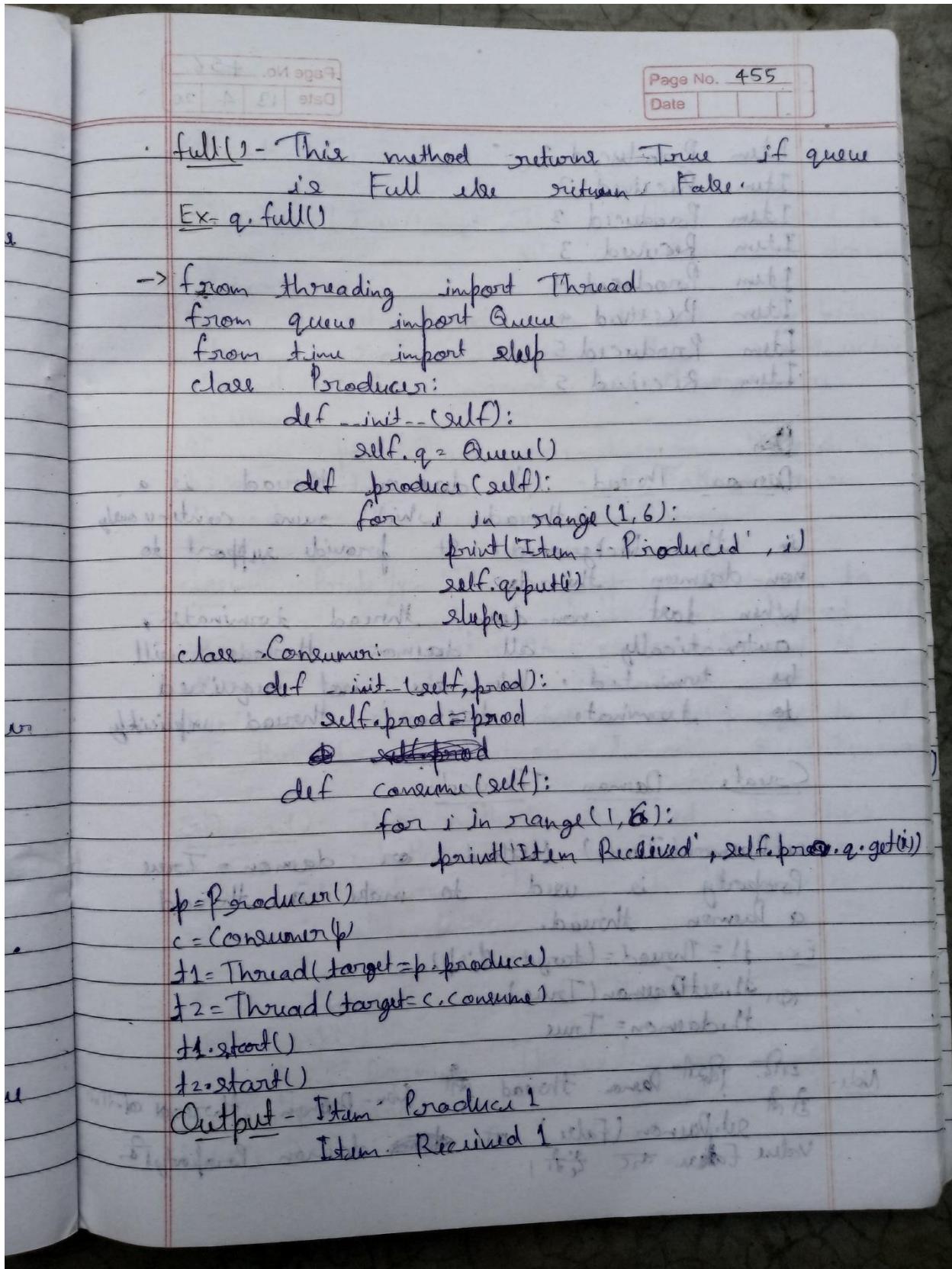
## Condition -

- Condition class is used to improve speed of communication between threads.
  - The condition class object is called condition variable.
  - A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default.  
Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.
  - A Condition is a more advanced version of the event object.









Page No. 456  
Date 13 4 20

Item Produced 2  
Item Received 2  
Item Produced 3  
Item Received 3  
Item Produced 4  
Item Received 4  
Item Produced 5  
Item Received 5

Daemon Thread - A daemon thread is a thread which runs continuously in the background. It provides support to non-daemon threads.

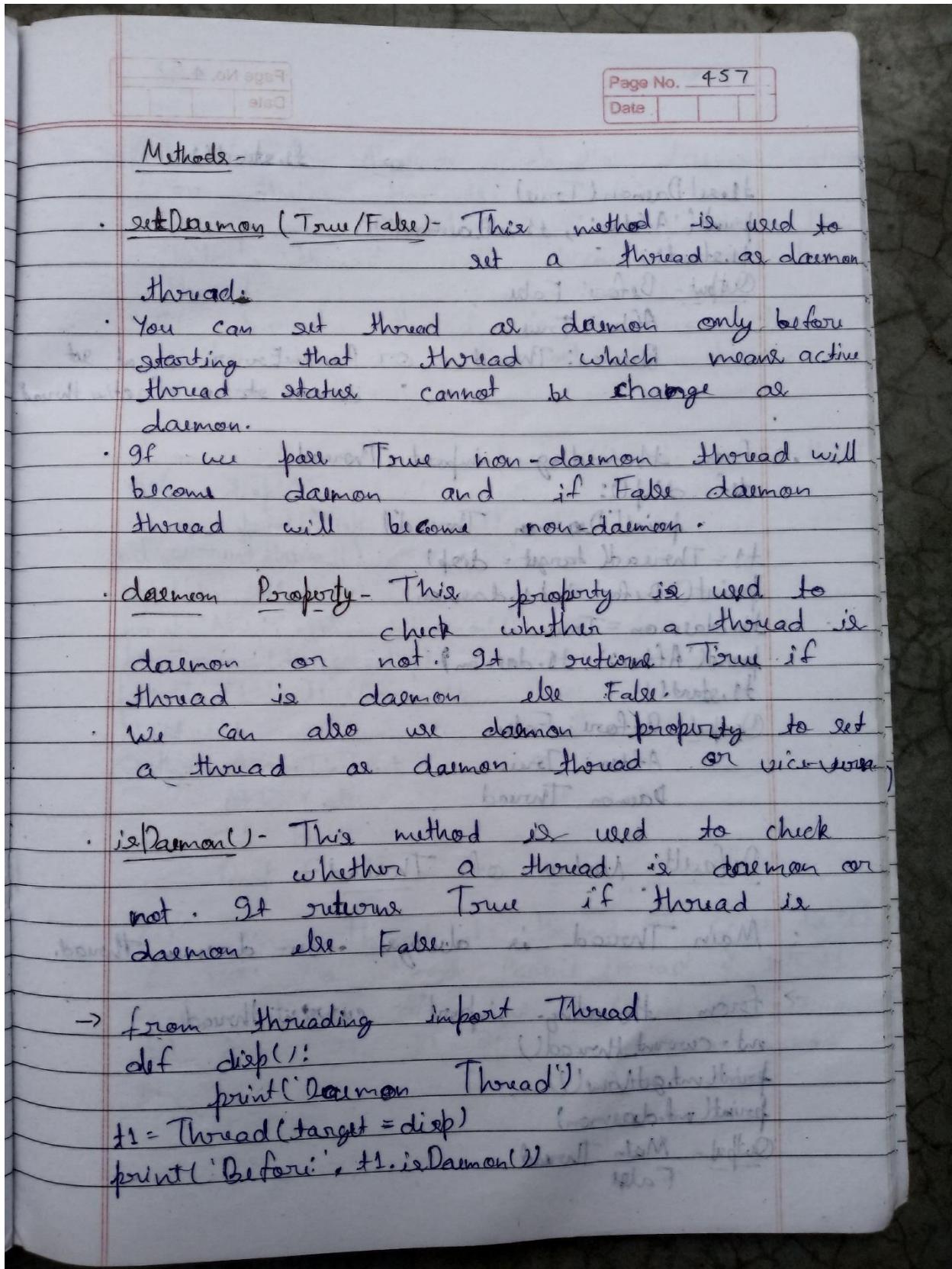
- When last non-daemon thread terminates, automatically all daemon threads will be terminated. We are not required to terminate daemon thread explicitly.

Create Daemon Thread -

setDaemon(True) Method or daemon = True Property is used to make a thread a daemon thread.

Ex - t1 = Thread(target=disk)  
or t1.setDaemon(True)  
t1.daemon = True

Note - If for a daemon thread it is Non-Daemon thread then setDaemon(False) or ~~not~~ daemon Property for value False on it.



Page No. 458  
Date

or `t1.start()`

`t1.setDaemon(True)`  
`print('After:', t1.isDaemon())`  
`t1.start()`

Output - Before: False  
After: True  
 Daemon: Thread or RuntimeError: cannot set daemon status of active thread

→ from threading import Thread  
`def disp():`  
 `print('Daemon Thread')`  
`t1 = Thread(target=disp)`  
`print('Before:', t1.daemon)`  
`t1.daemon = True`  
`print('After:', t1.daemon)`  
`t1.start()`

Output - Before: False  
After: True  
 Daemon Thread

Default: Nature of Thread -

- Main Thread is always non-daemon thread.

→ from threading import current\_thread  
`mt = current_thread()`  
`print(mt.getName())`  
`print(mt.isDaemon())`

Output - Main Thread: False

Page No. 459  
Date

- Rest of the threads inherit daemon nature from their parents.
- If parent thread is non-daemon then child thread will become non-daemon thread.
- If parent thread is daemon then child thread will also become a daemon thread.

→ from threading import Thread, current\_thread

```

def disp():
    print('Disp Function')
mt = current_thread()
print(mt.getName())
print('MT:', mt.isDaemon())
t1 = Thread(target=disp)
print('T1:', t1.isDaemon())
t1.start()

```

Output - Main Thread

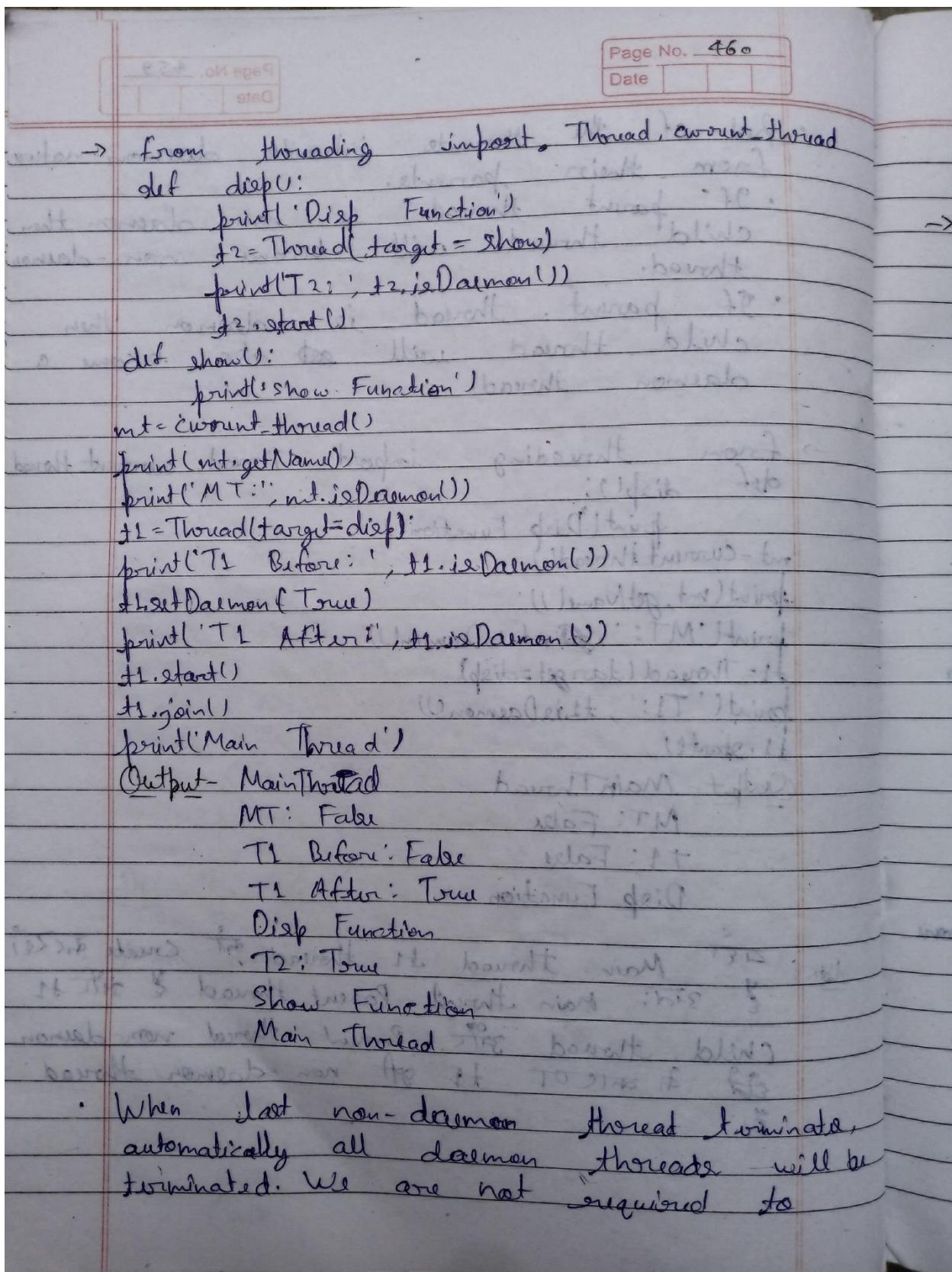
```

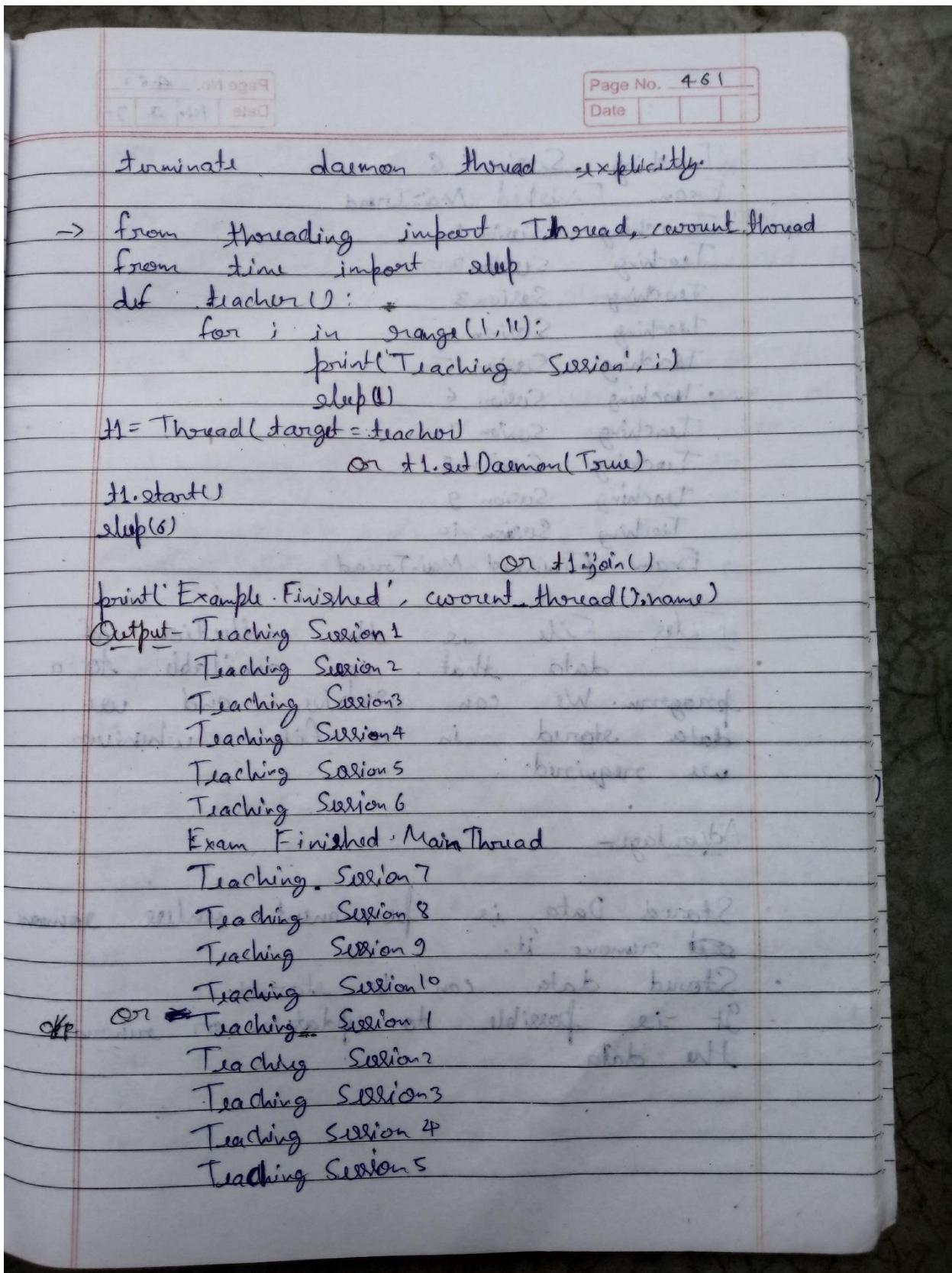
MT: False
T1: False
Disp Function

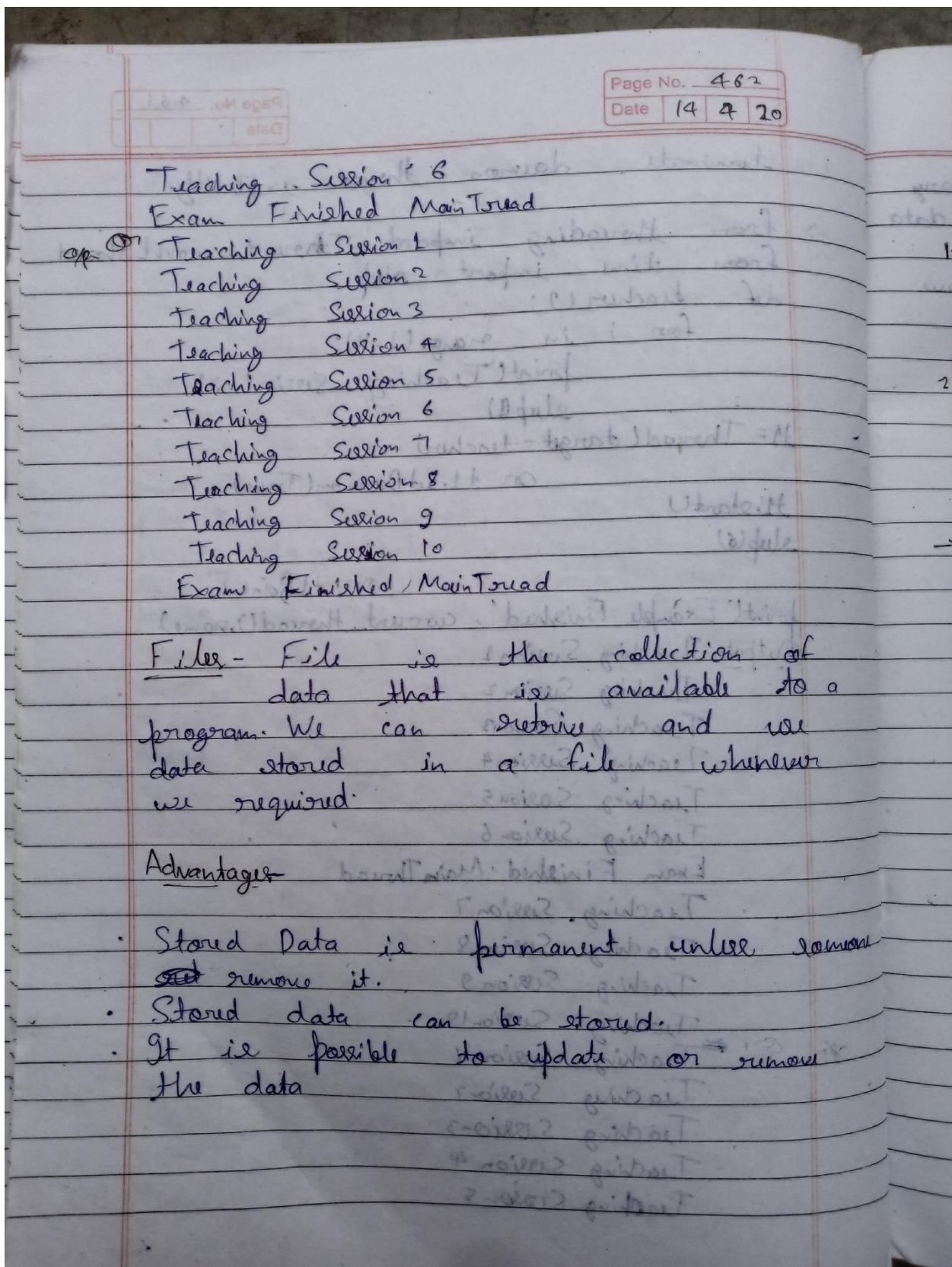
```

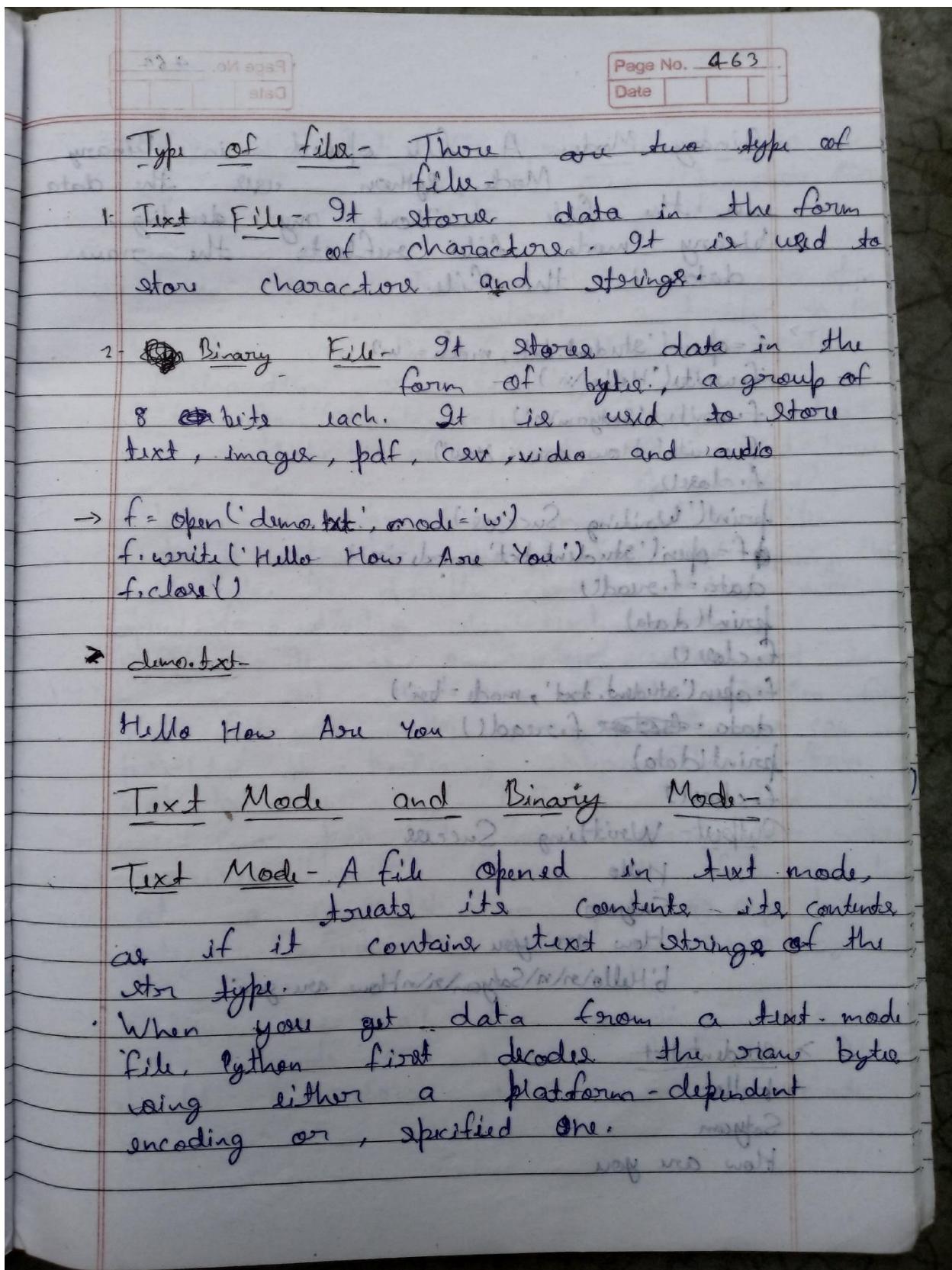
Note:

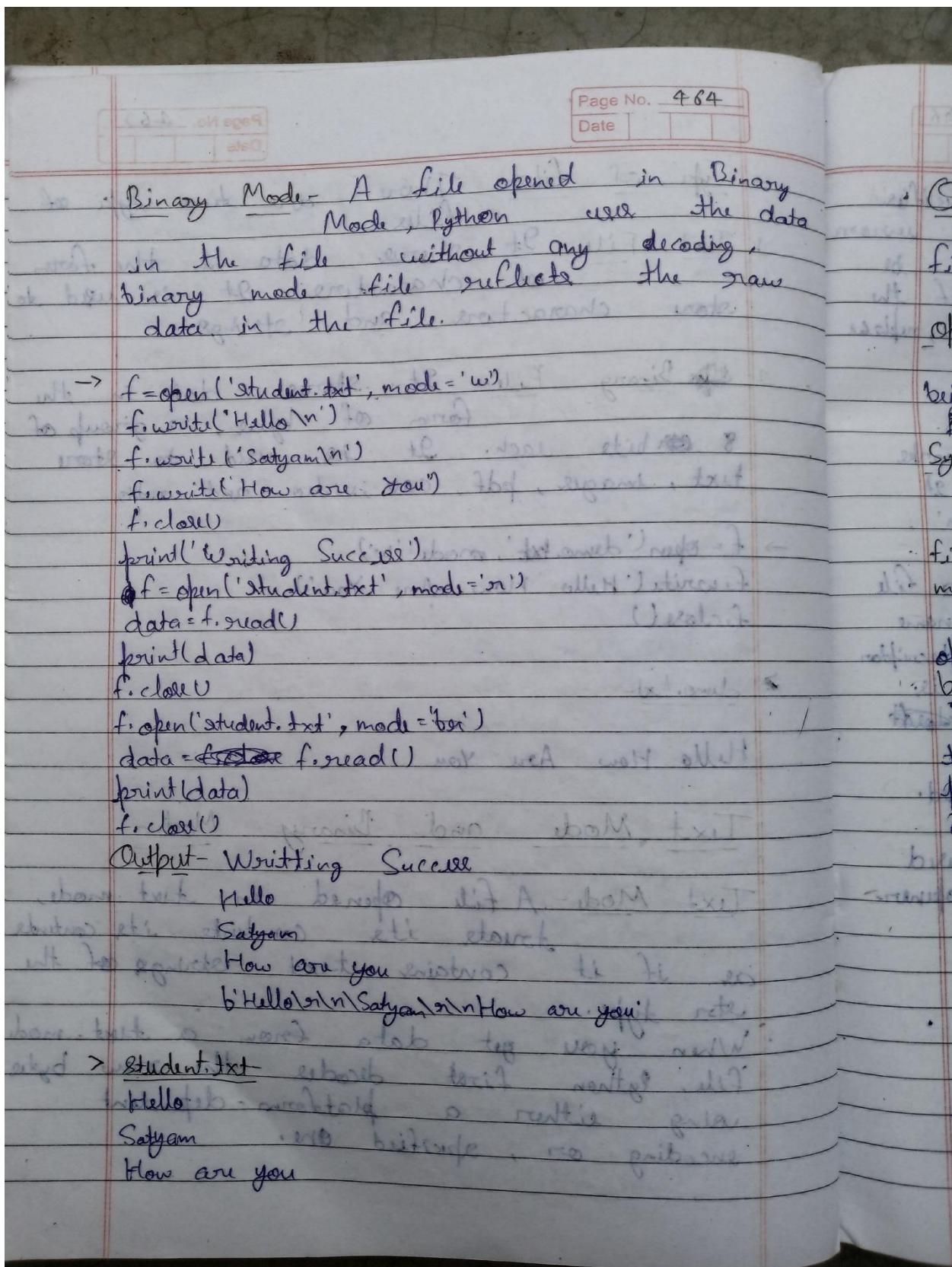
>Main thread is thread. It creates all other threads.  
 Main thread is Parent thread & 3rd t1  
 Child thread 3rd Parent thread non-daemon  
 An instance of t1 is non-daemon thread

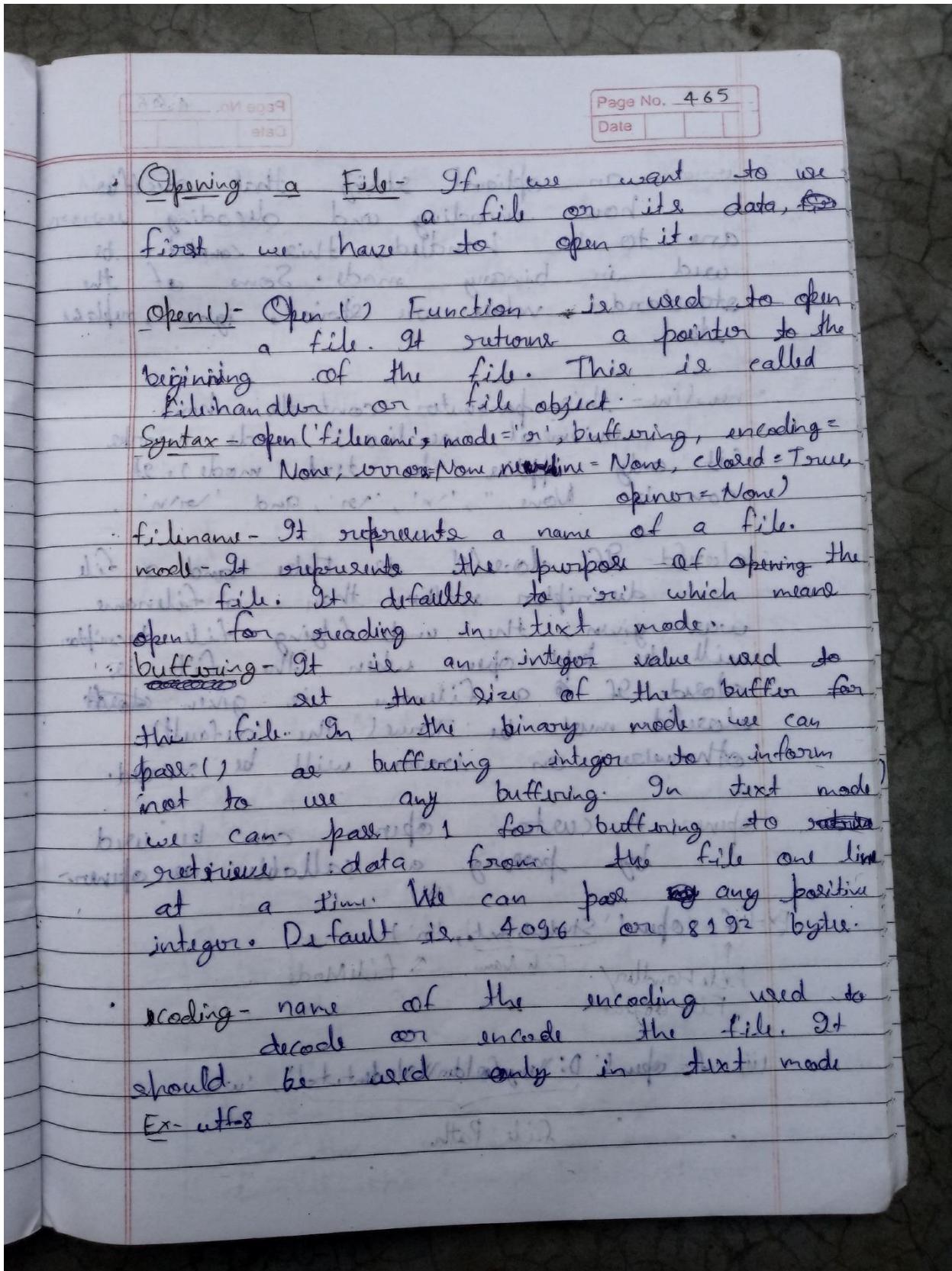


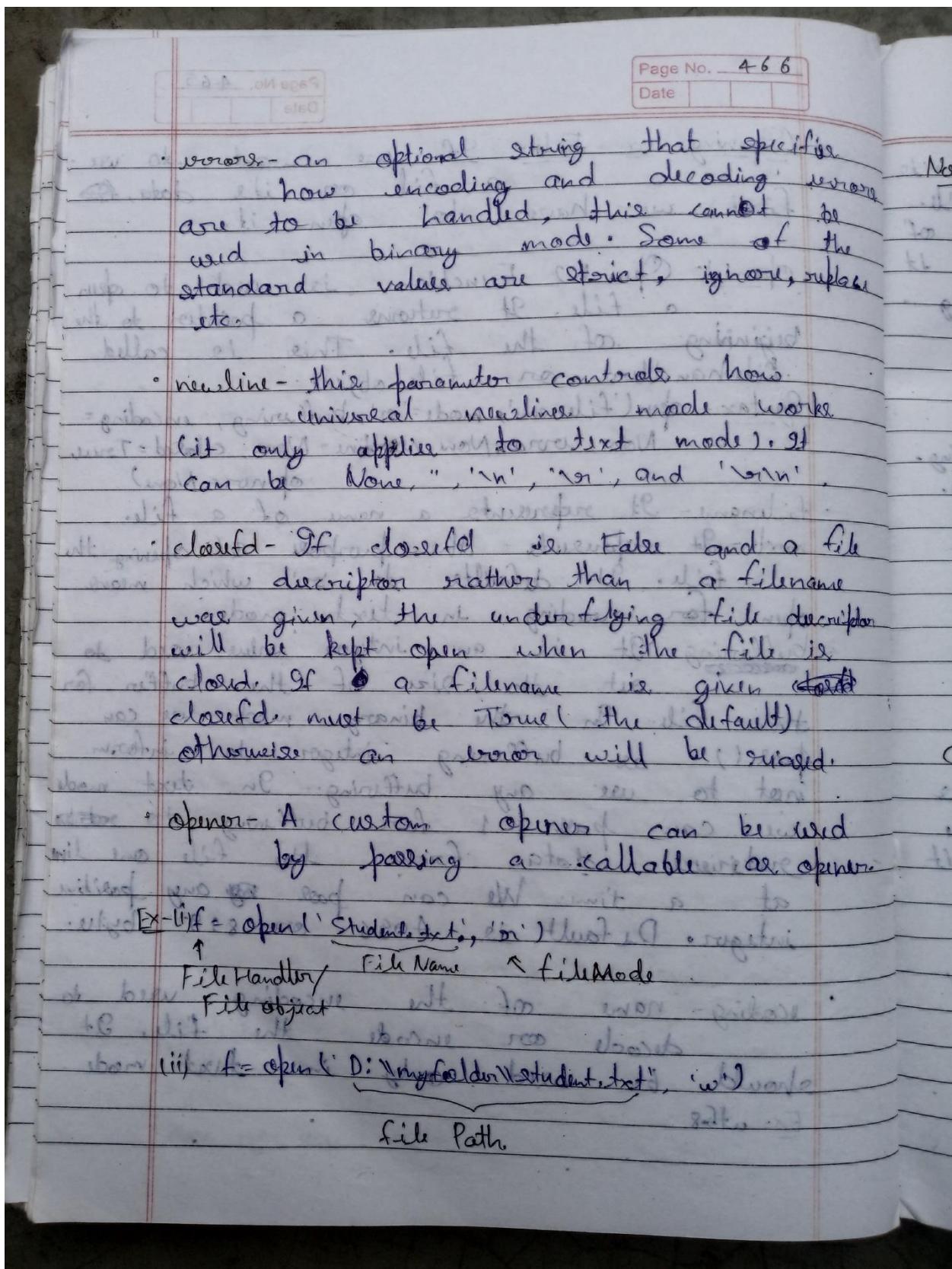


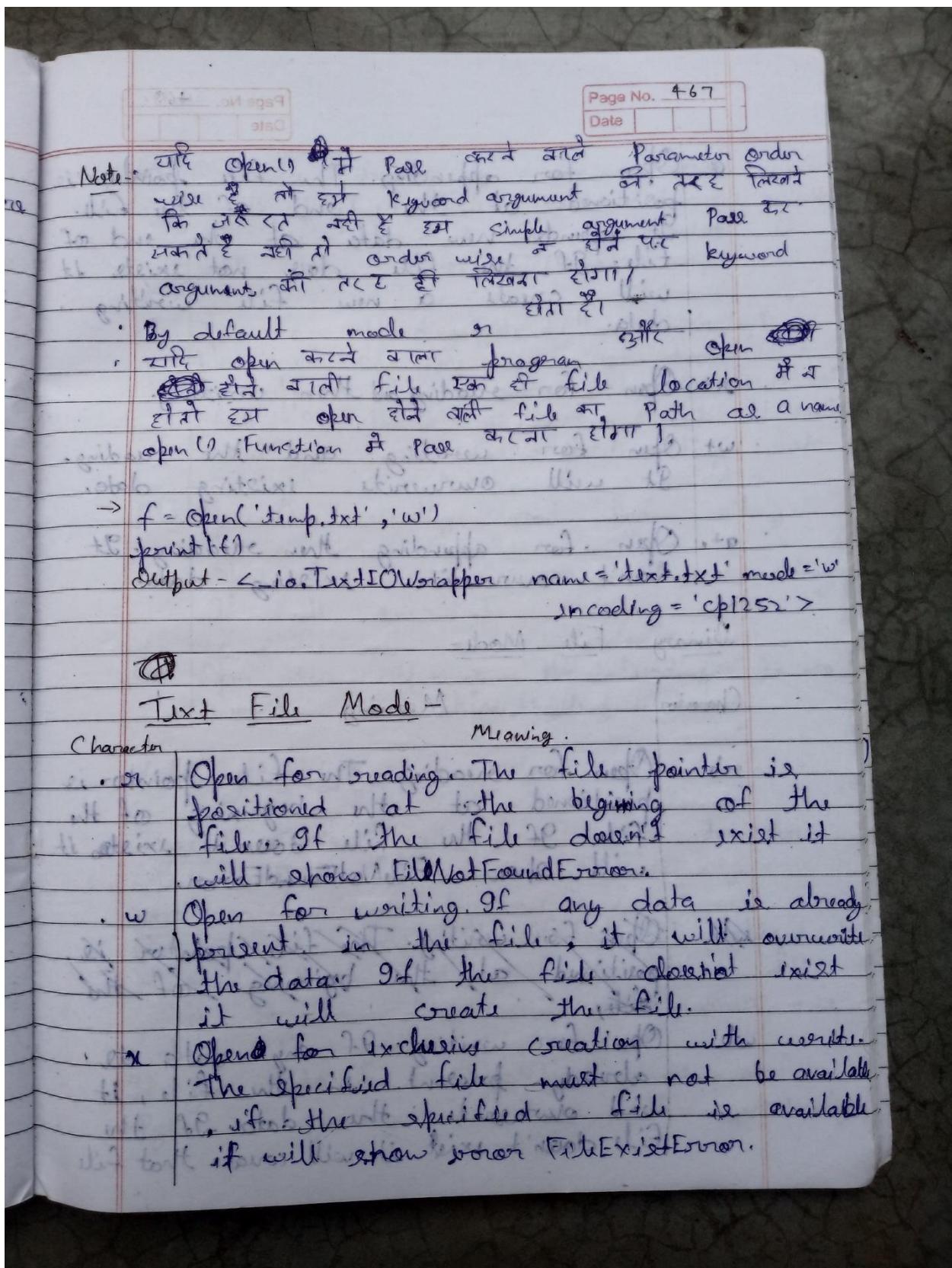




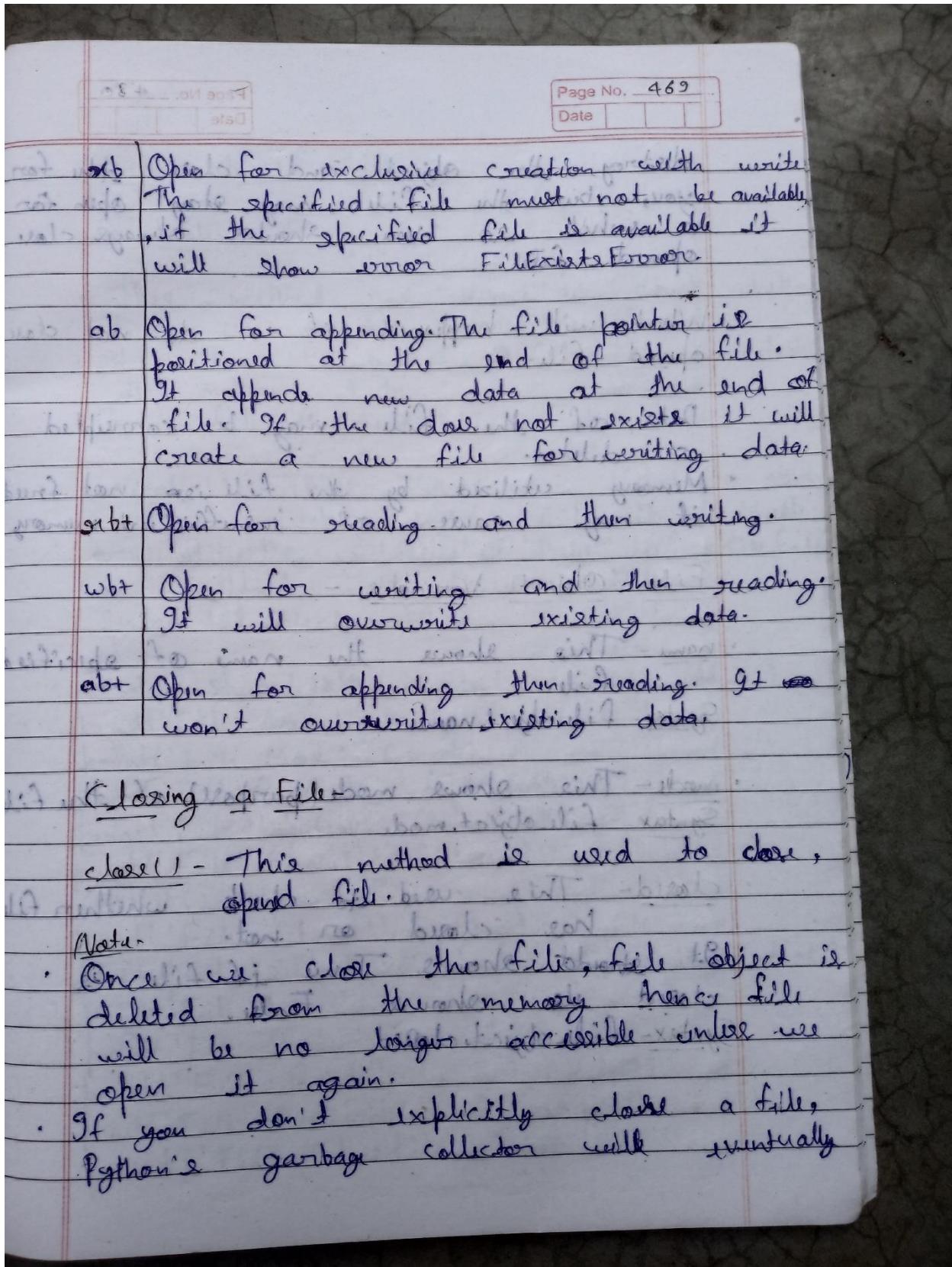


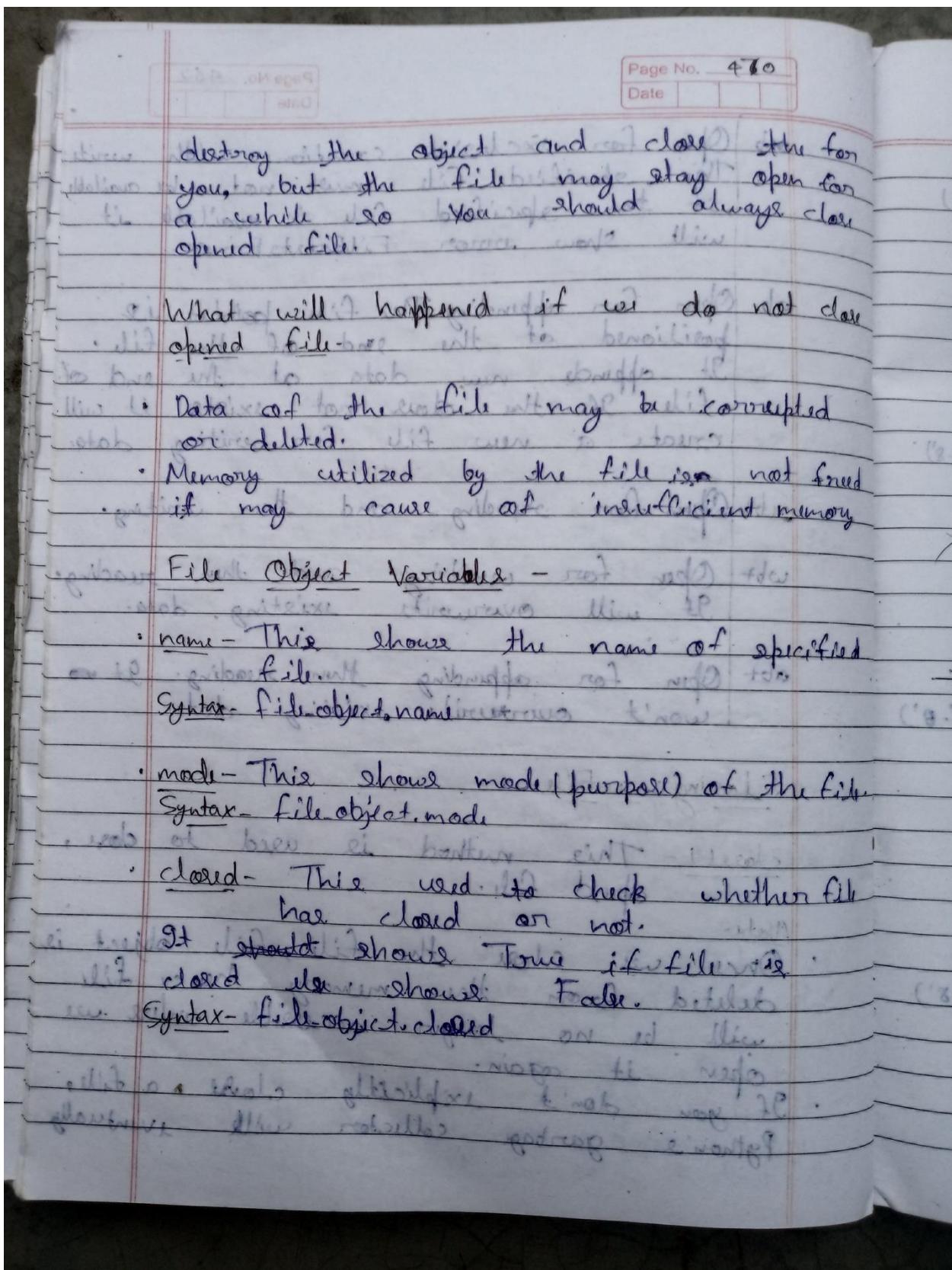


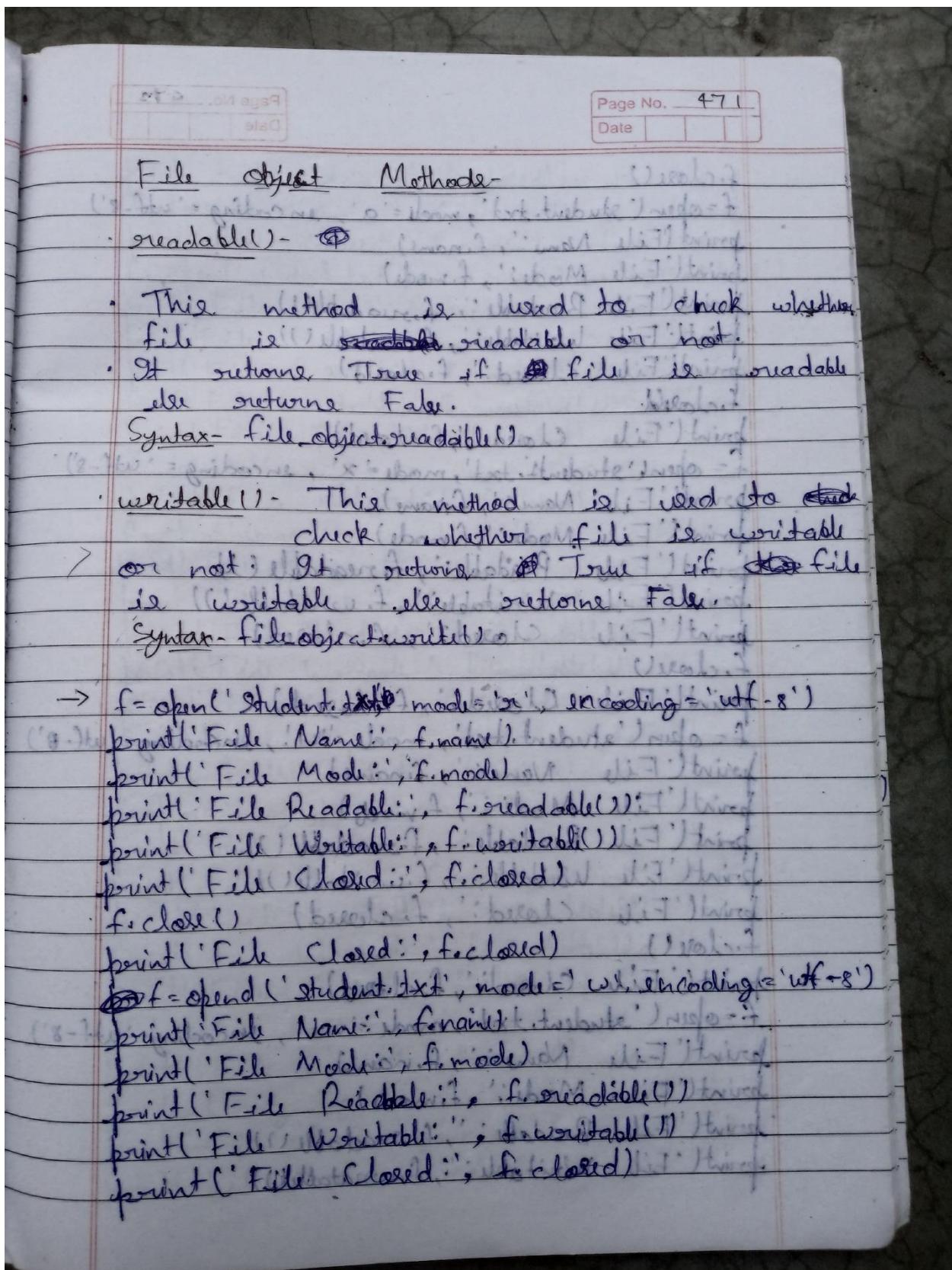


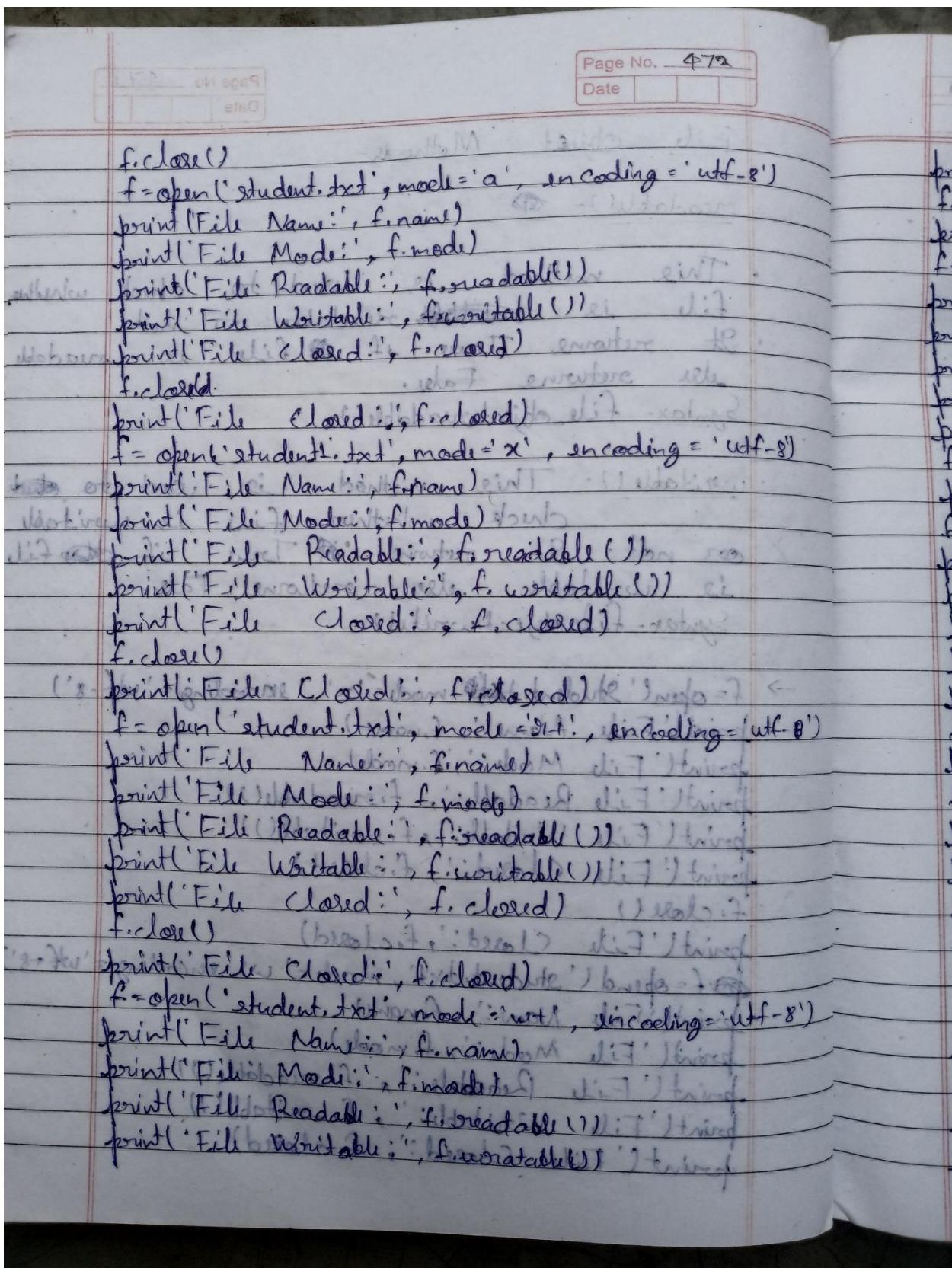


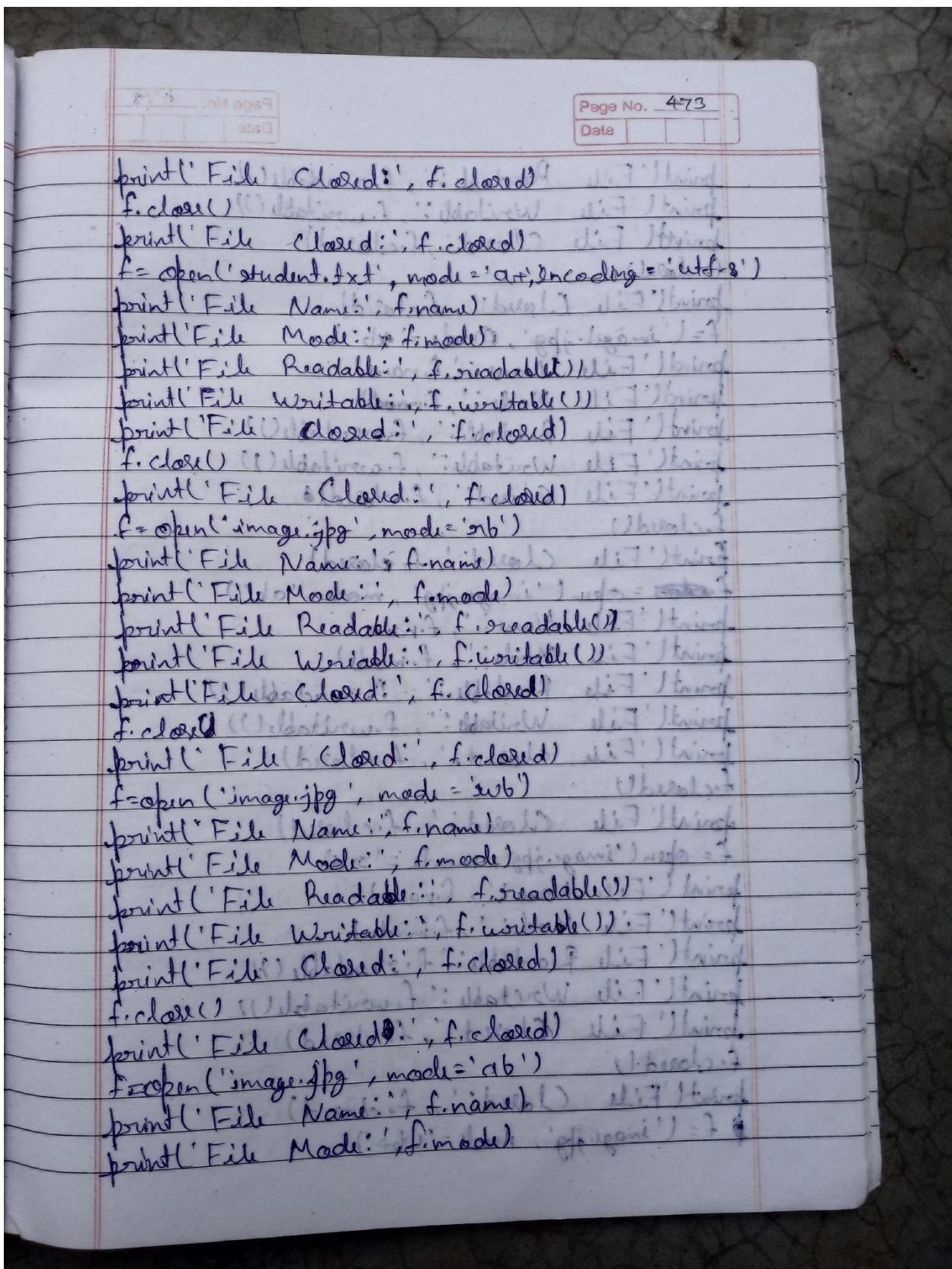
		Page No. 468	Date
a	Open for appending. The file pointer is positioned at the end of the file. It appends new data at the end of file. If the file does not exist, it will create a new file containing data.		
r+	Open for reading and then writing.		
wt	Open for writing and then reading. It will overwrite existing data.		
at	Open for appending then reading. It won't overwrite existing data.		
<u>Binary File Mode-</u>			
Character	Meaning		
rb	Open for reading. The file pointer is positioned at the beginning of the file. If the file doesn't exist, it will show FileNotFoundError.		
wb	Open for writing. The file pointer is positioned at the beginning of the file. If any data is already present in the file, it will overwrite the data. If the file doesn't exist, it will create that file.		



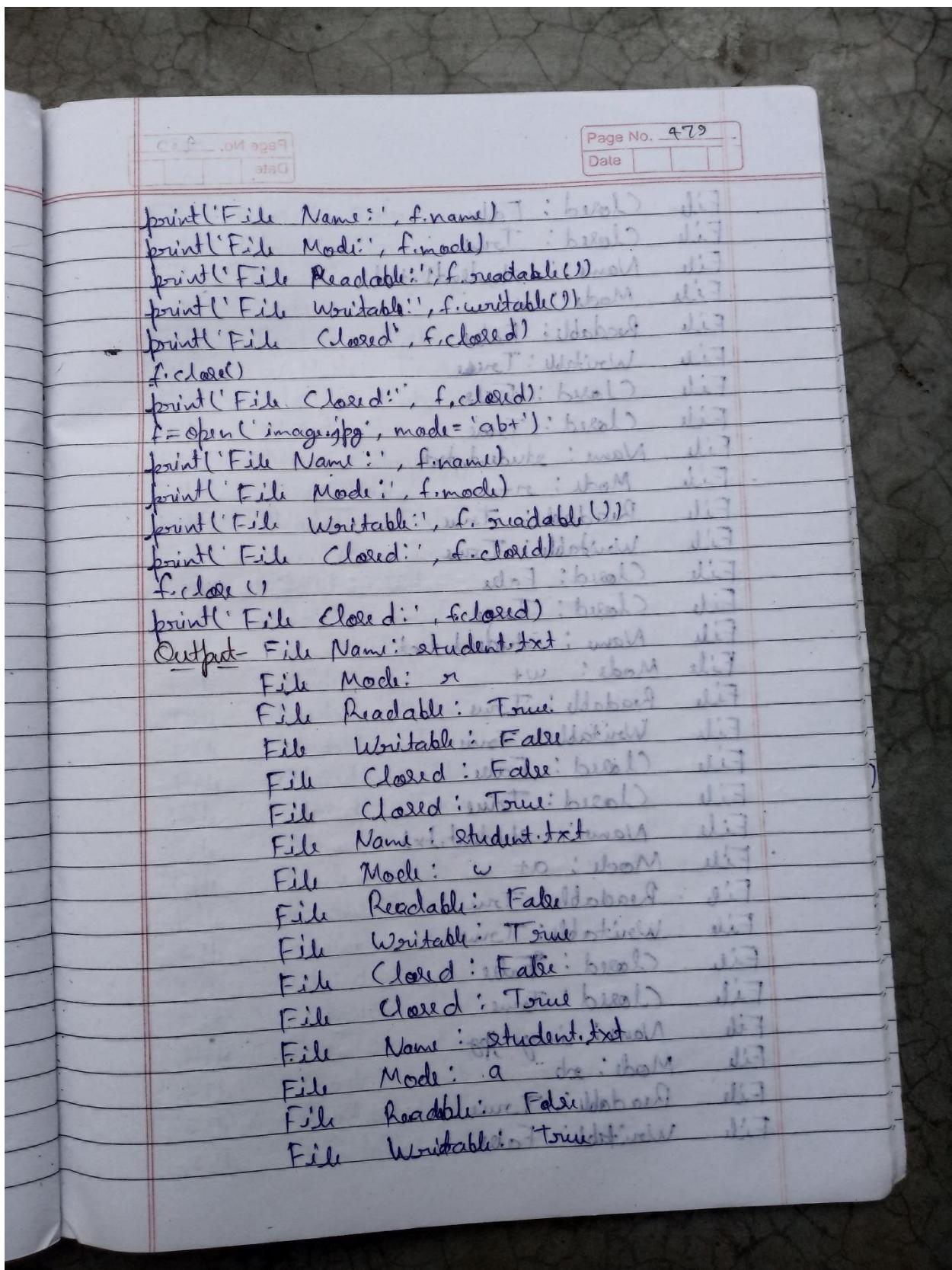


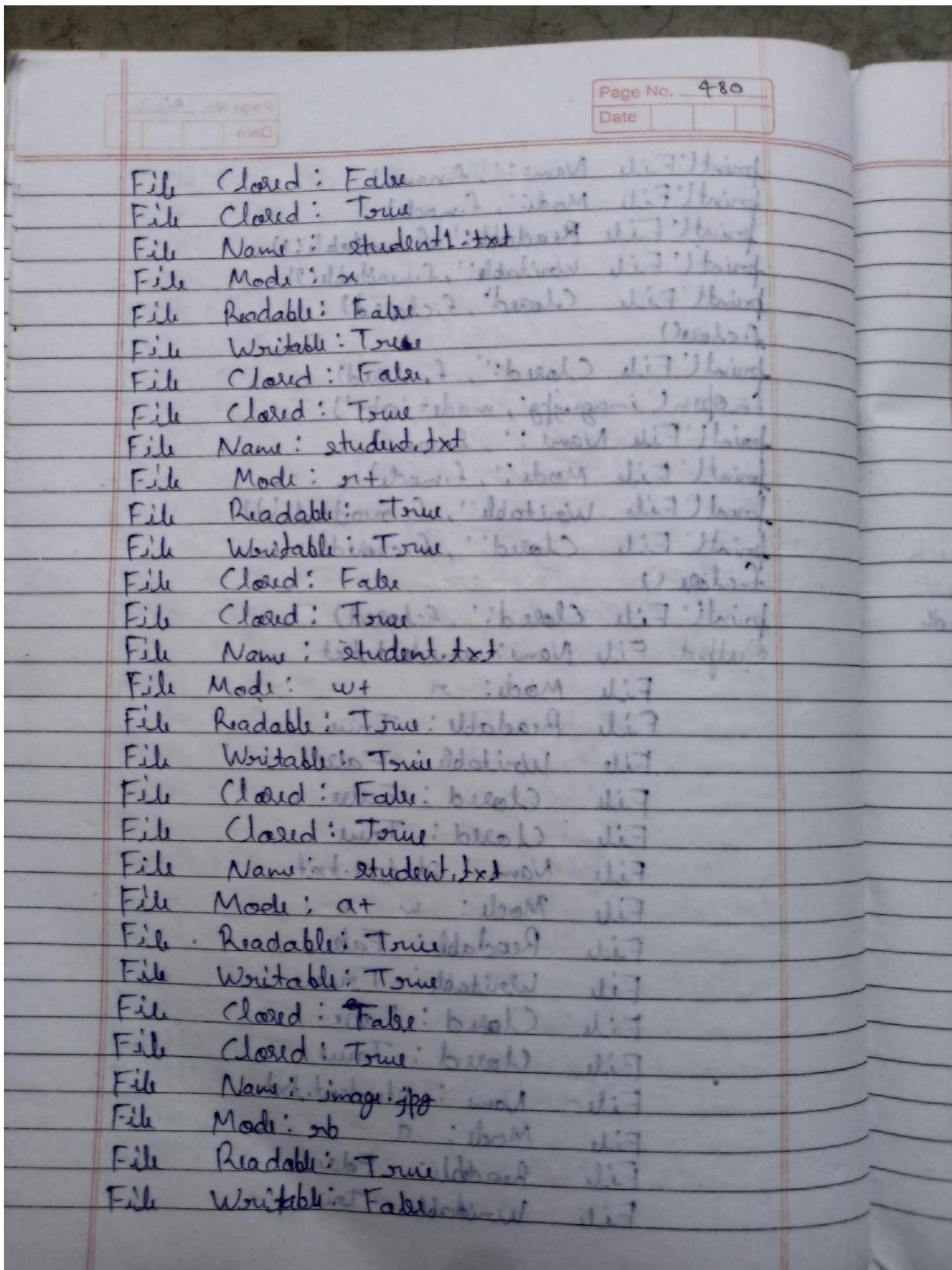


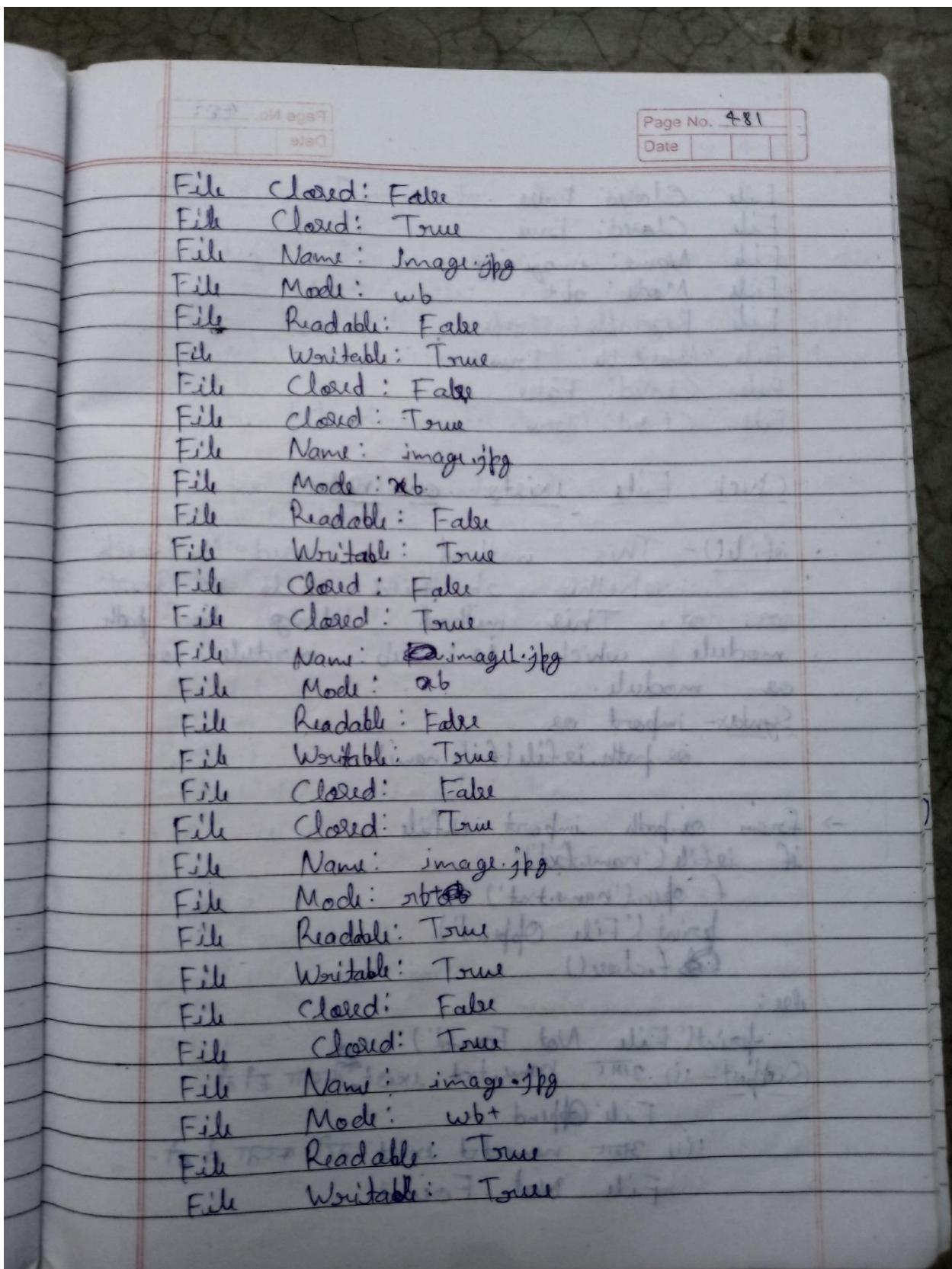












Page No. 482  
Date

File Closed: False  
 File Closed: True  
 File Name: image.jpg  
 File Mode: ab+  
 File Readable: True  
 File Writable: True  
 File Closed: False  
 File Closed: True

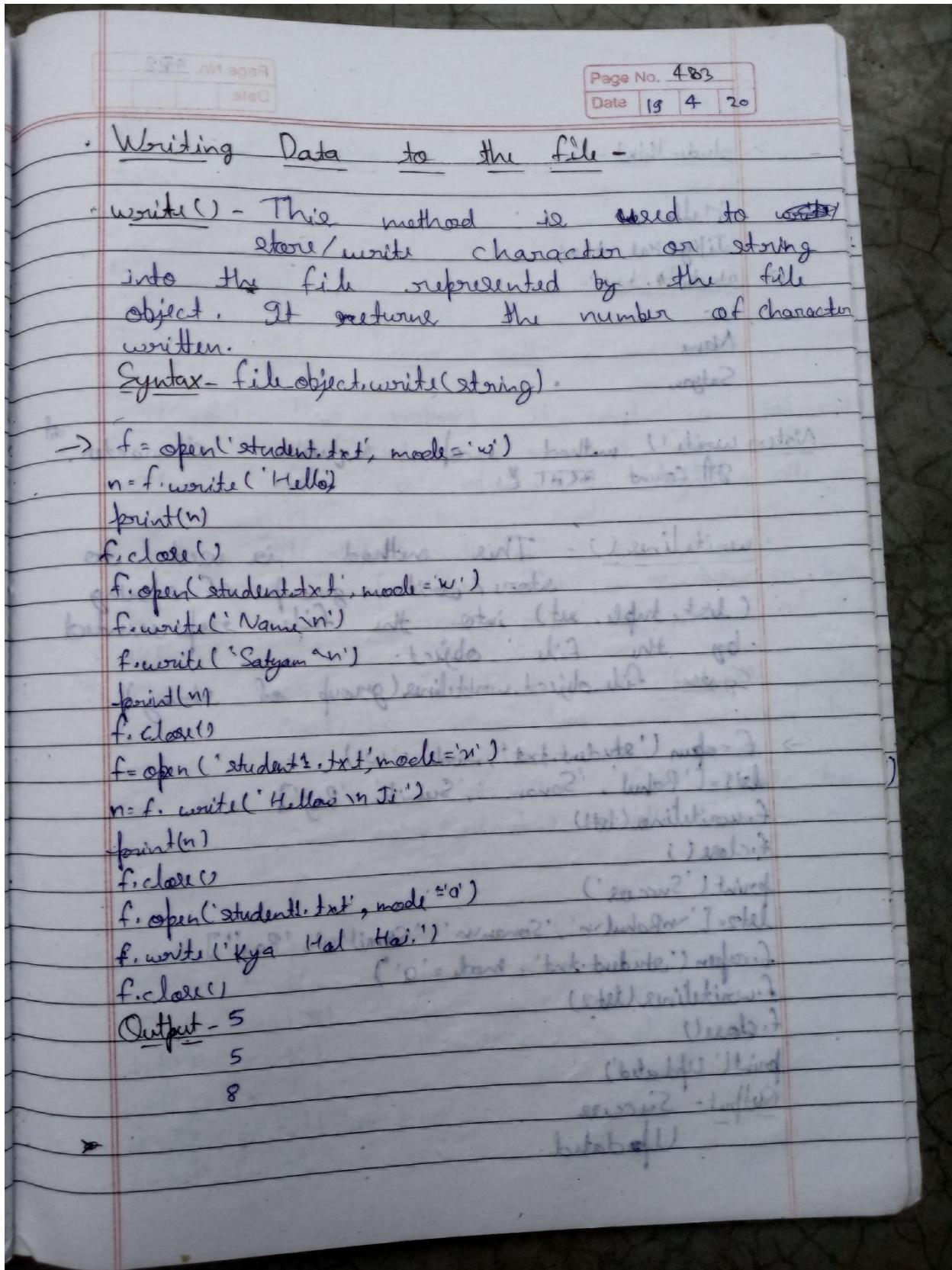
Check File exists or not - →

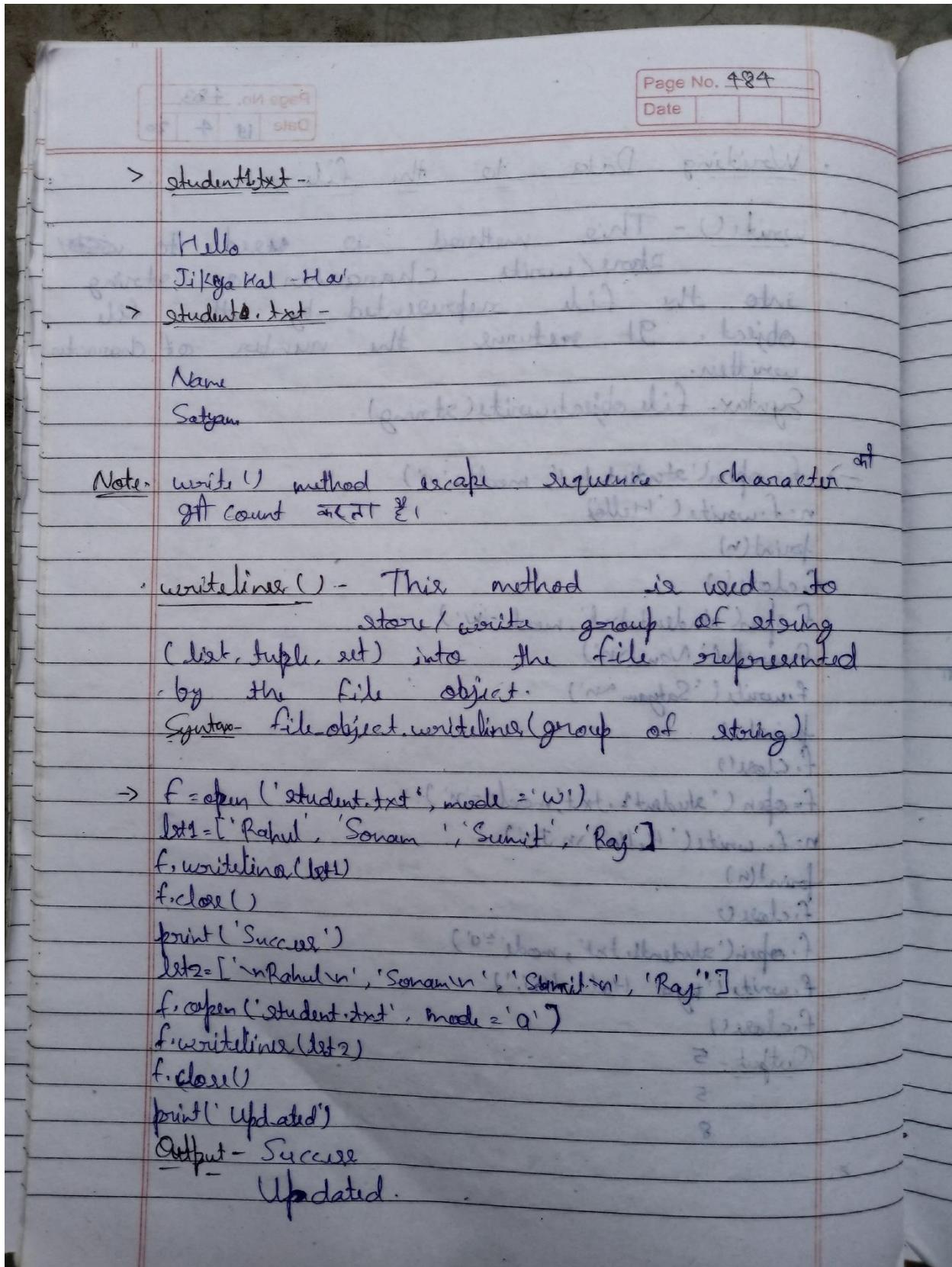
- isfile() - This method is used to check whether specified file is exists or not. This method belongs to path module which is sub module of os module.

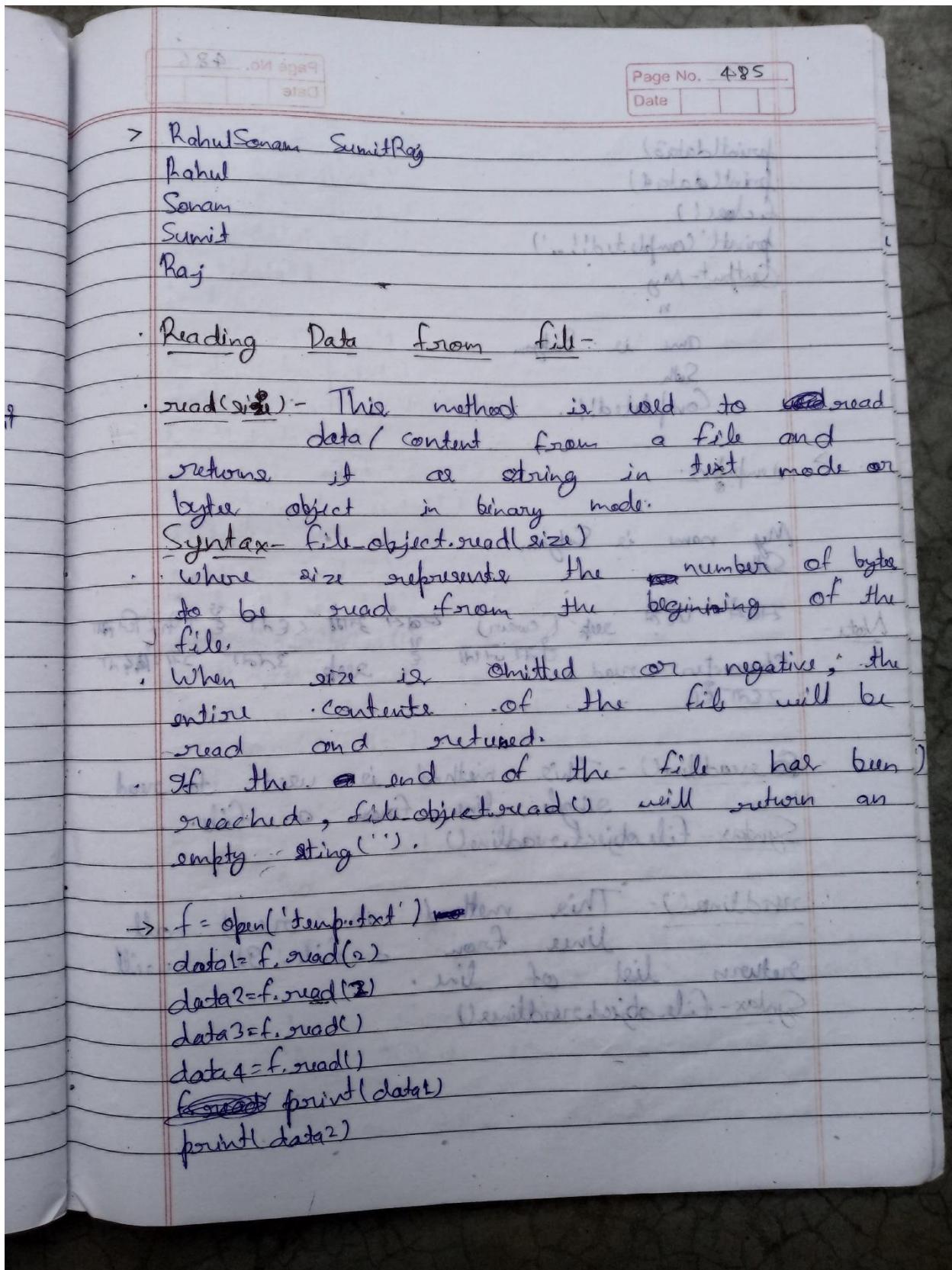
Syntax - import os  
os.path.isfile(file\_name)

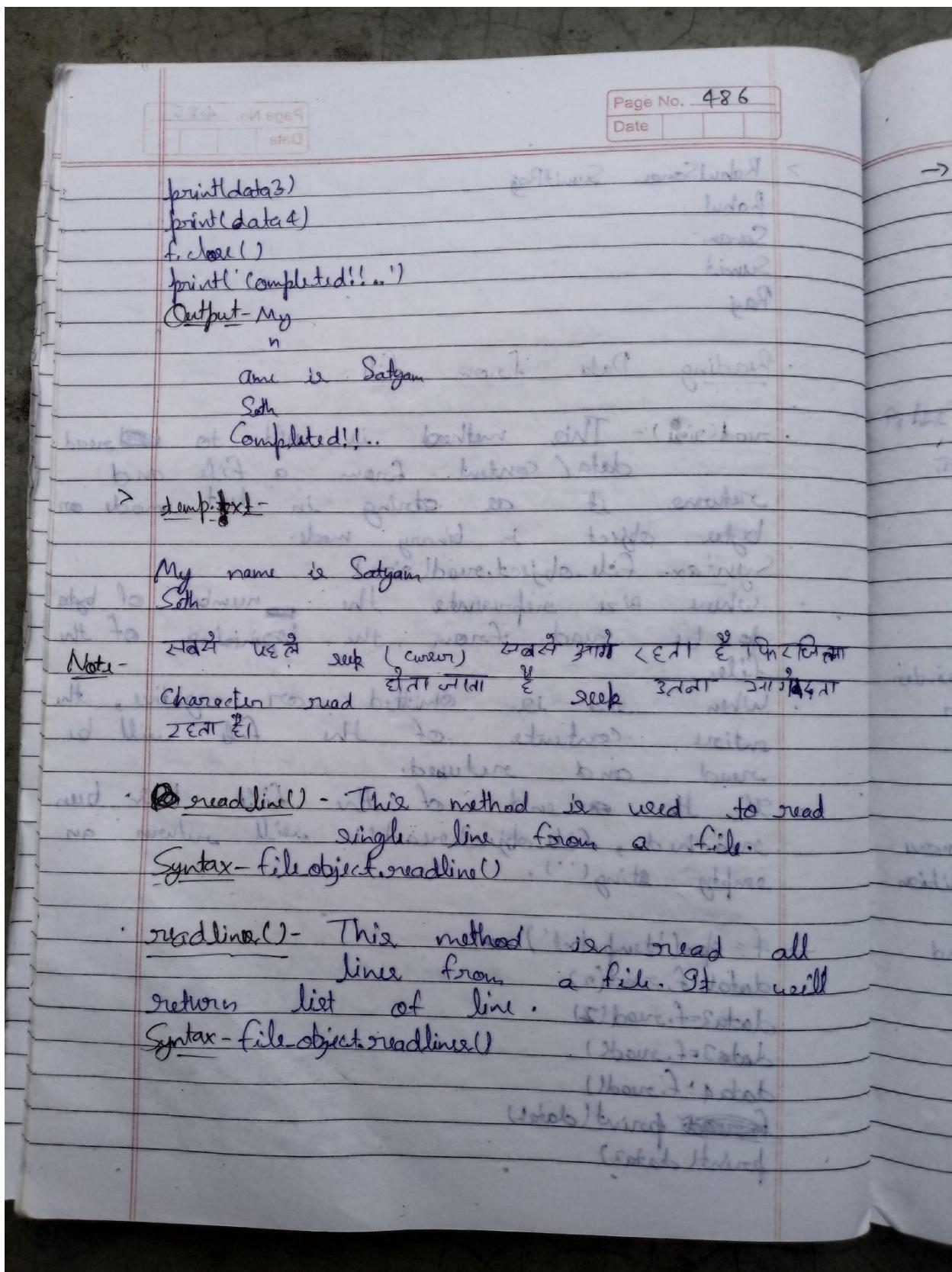
→ from os.path import isfile  
if isfile('name.txt'):  
 f = open('name.txt')  
 print('File Opened')  
 f.close()  
else:  
 print('File Not Found')

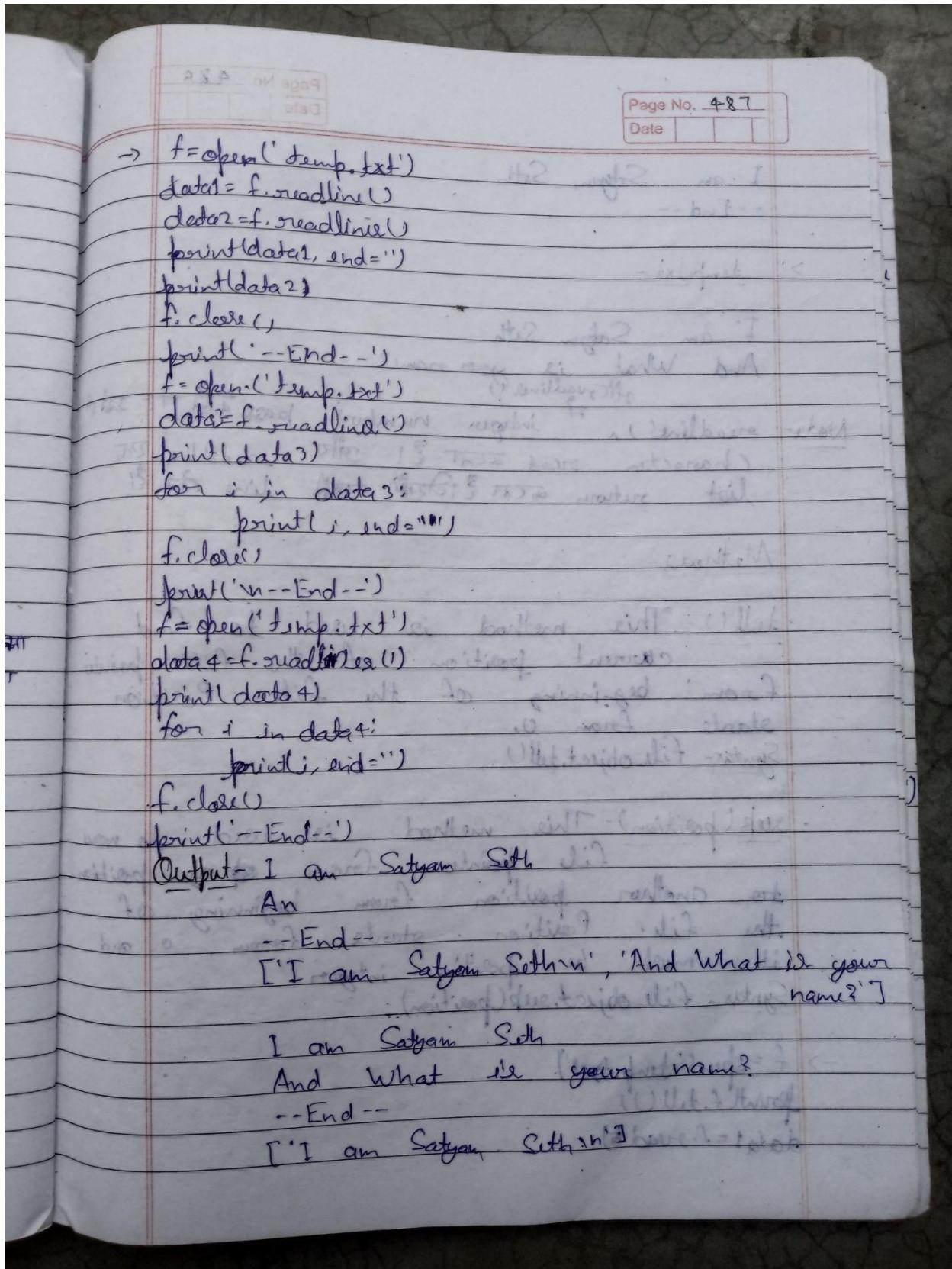
Output - (i) name.txt exist ~~so it will~~ -  
File Opened  
(ii) name.txt not exist ~~so it will~~ -  
File Not Found

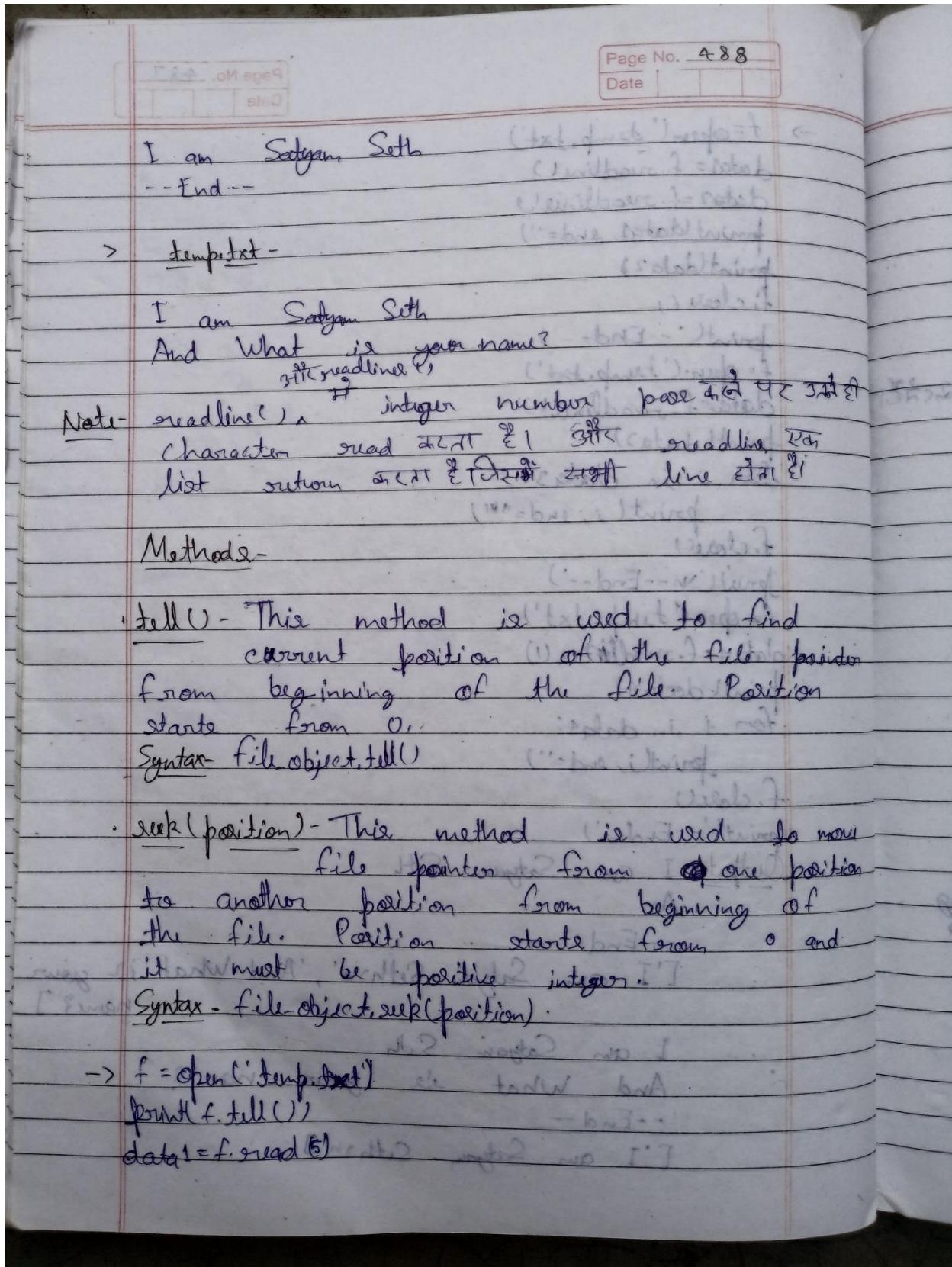


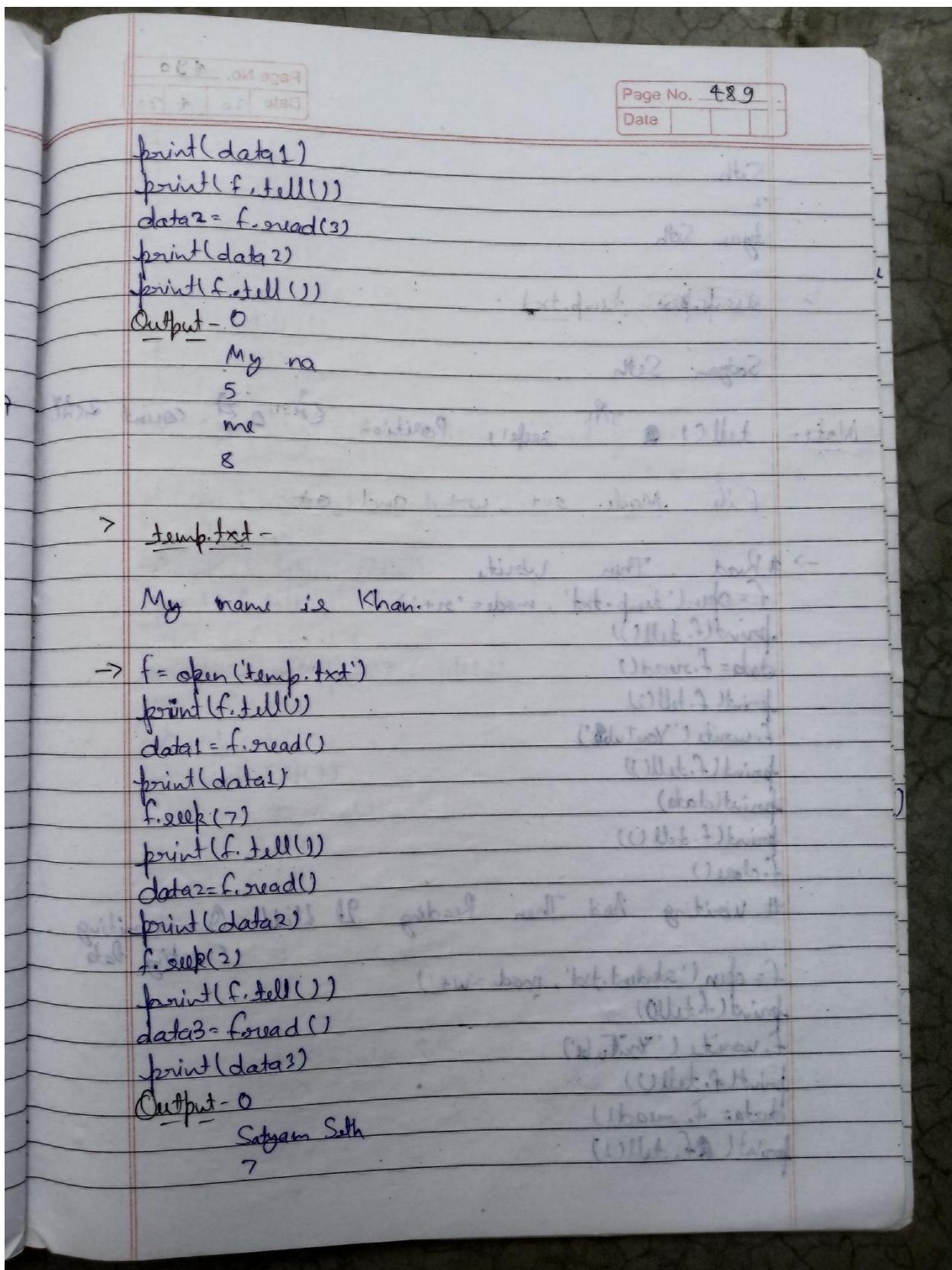


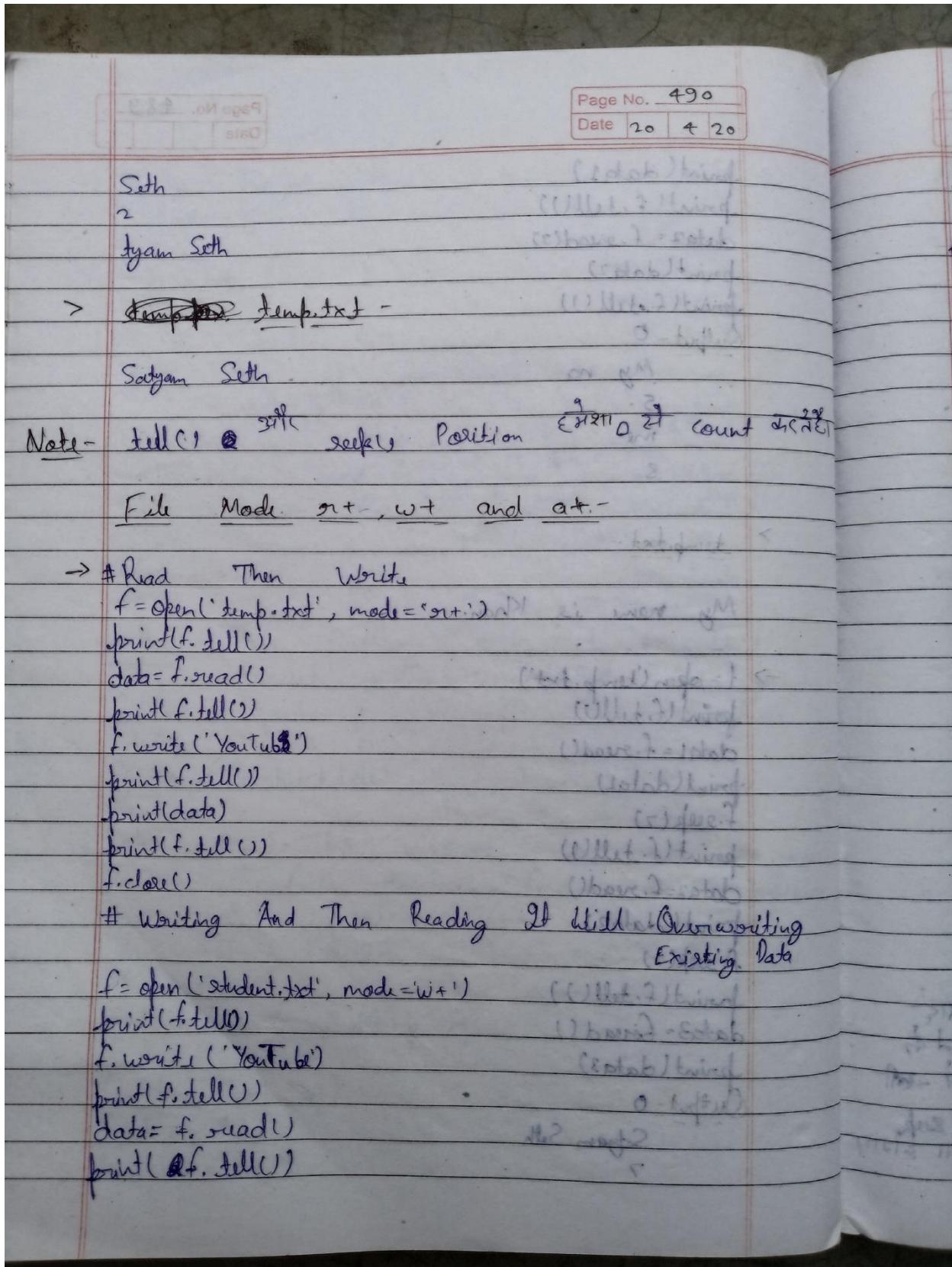


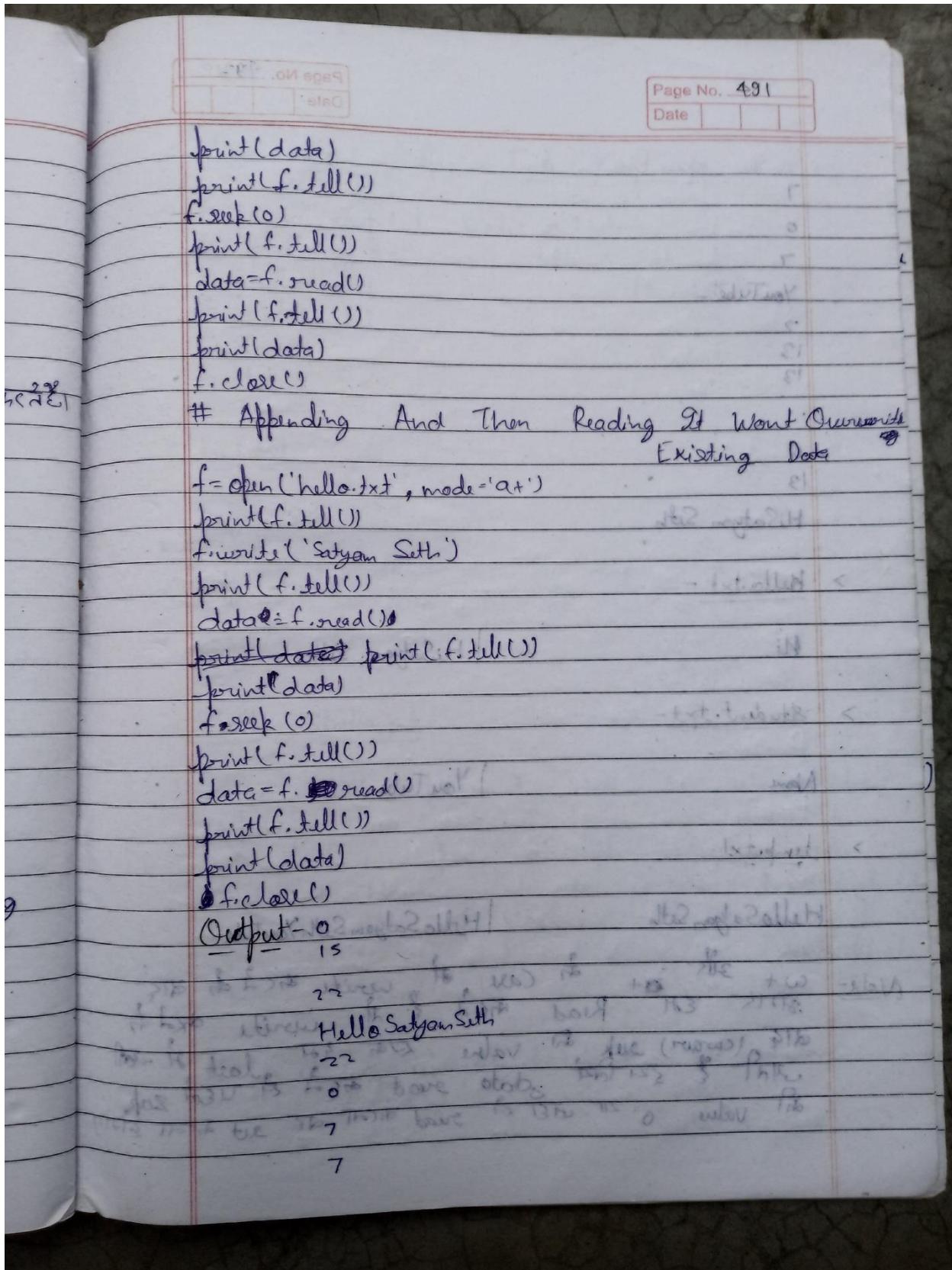


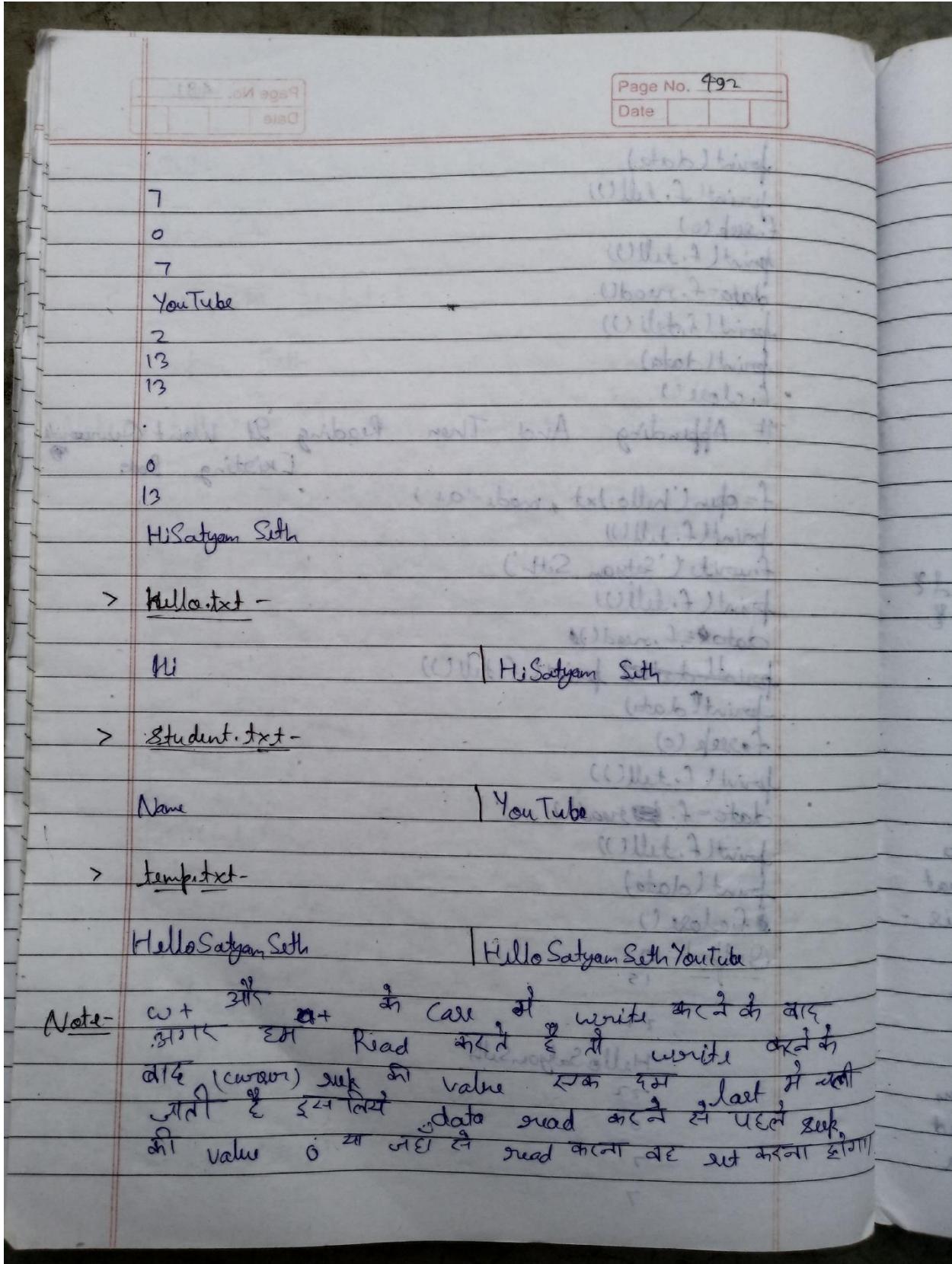


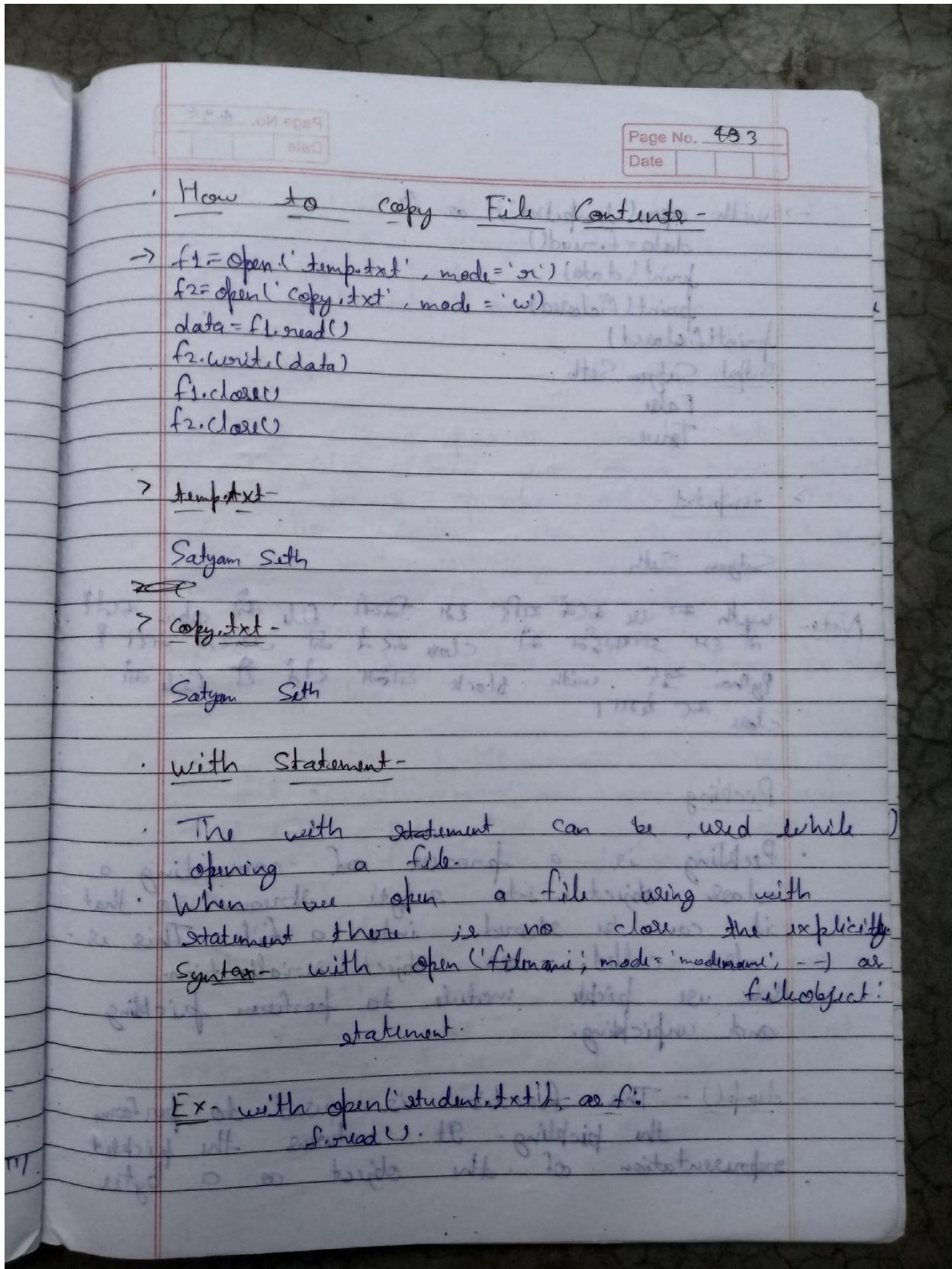


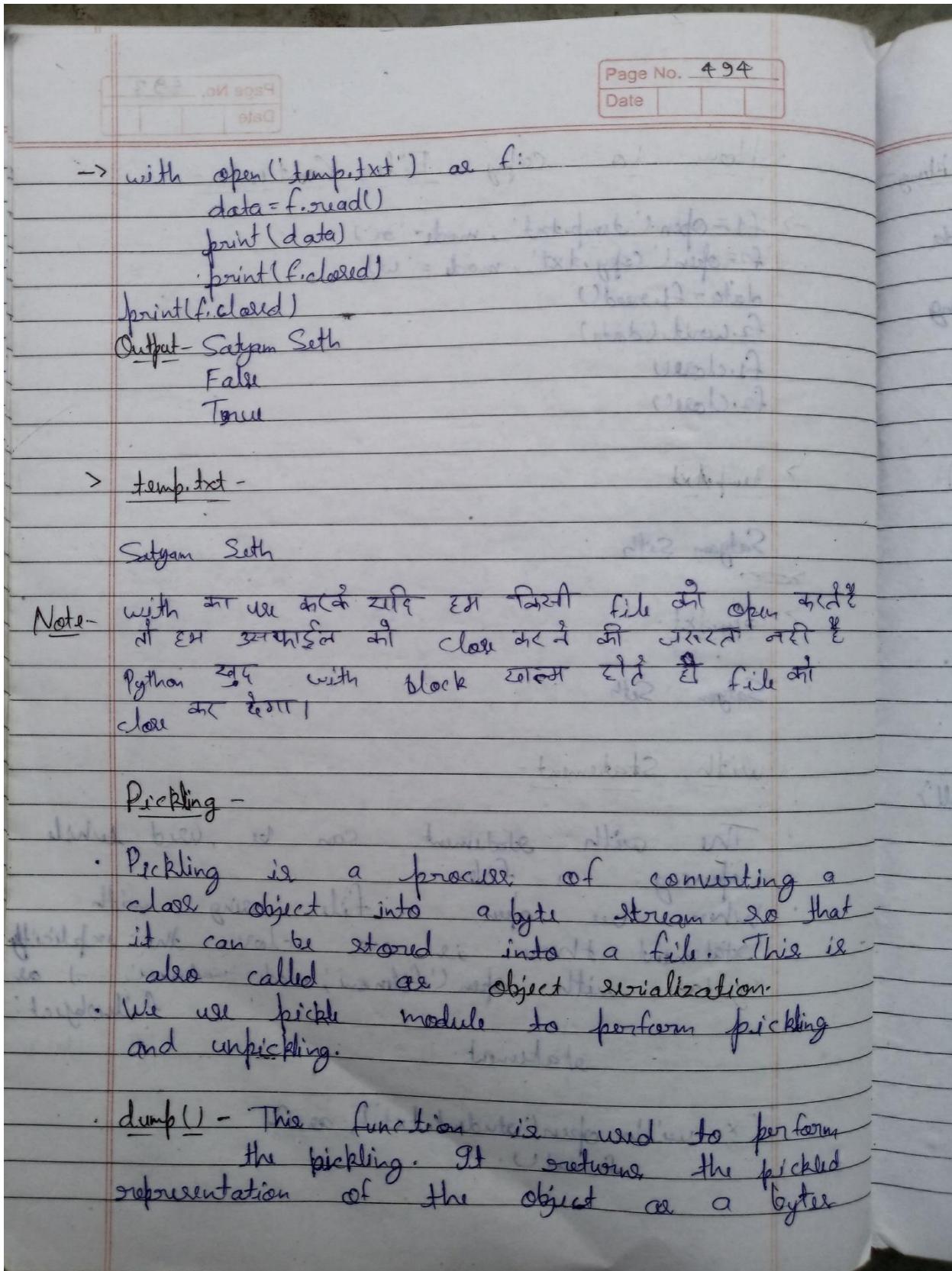


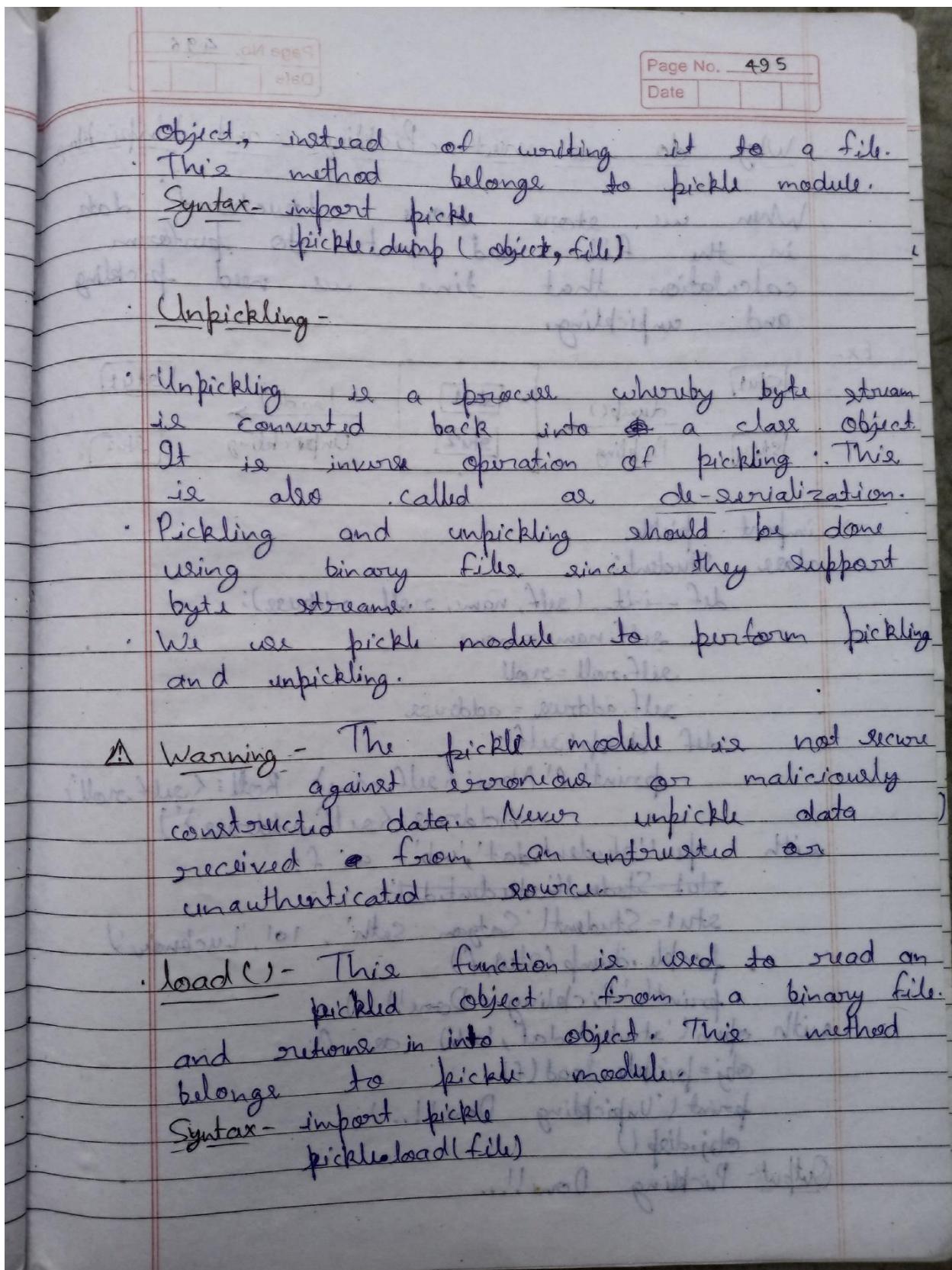












Page No. 496  
Date

Why do we need Pickling and Unpickling

When we store some structured data in the file and want to perform calculation that time we need pickling and unpickling.

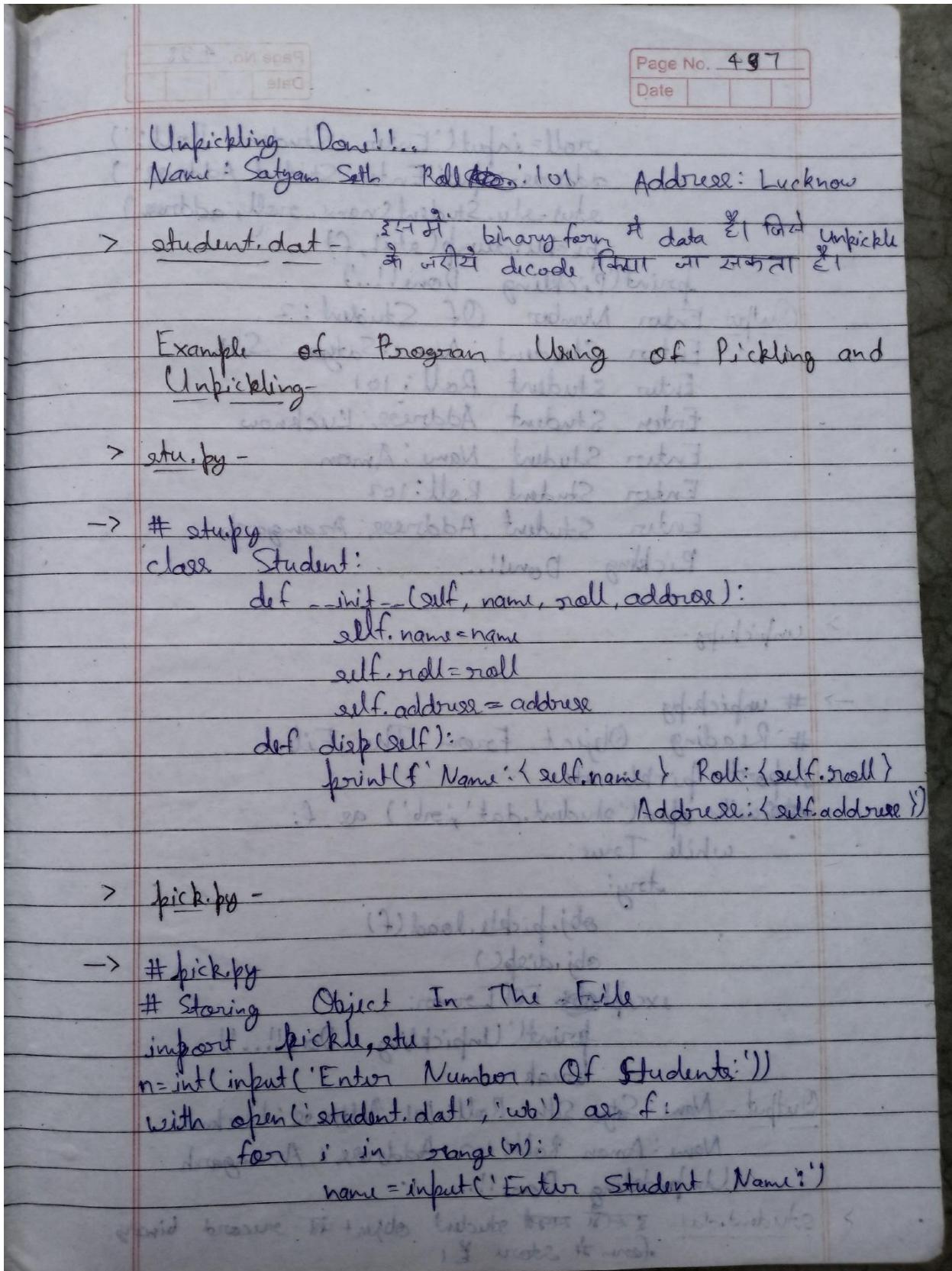
Ex-

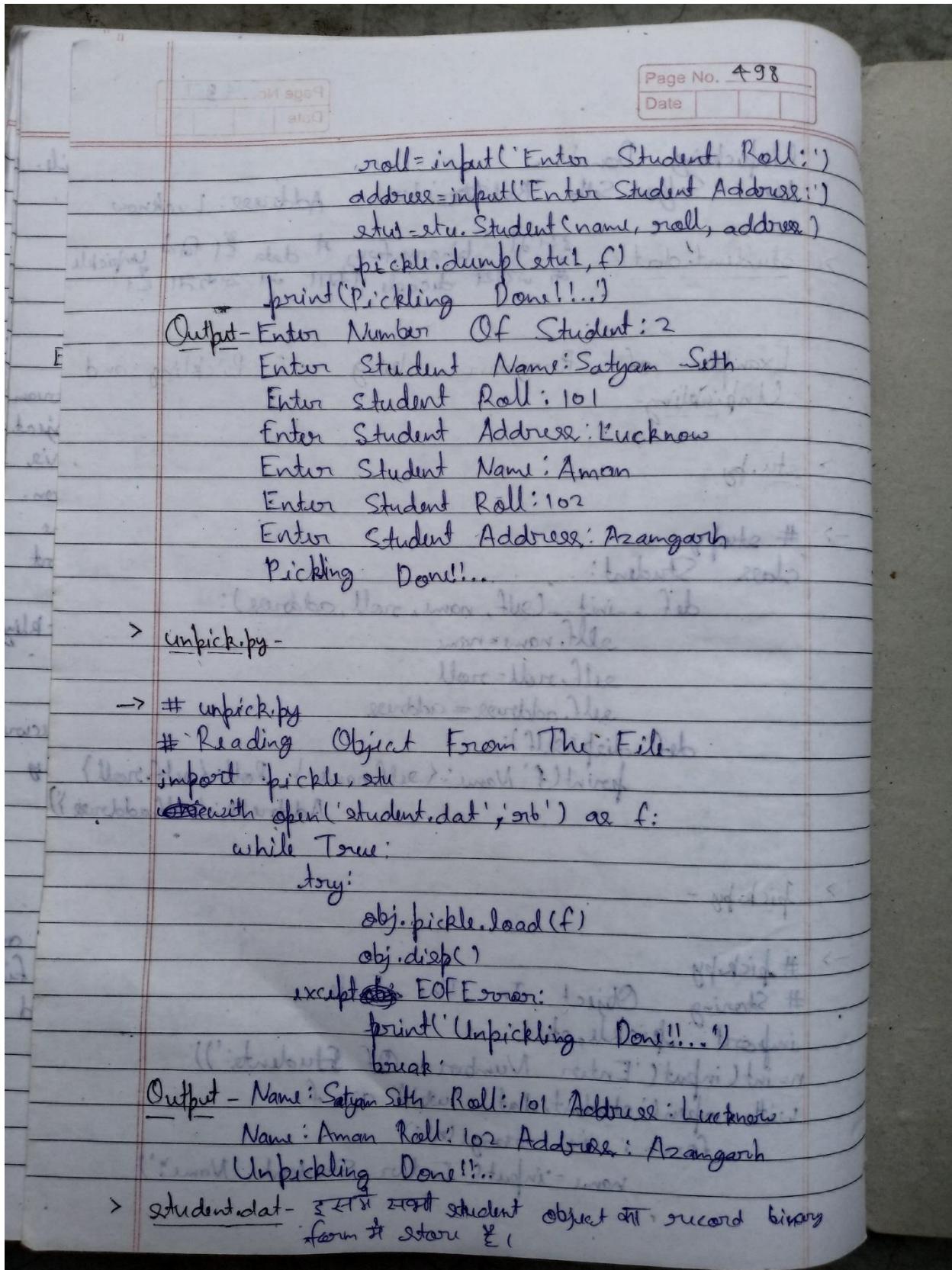
```

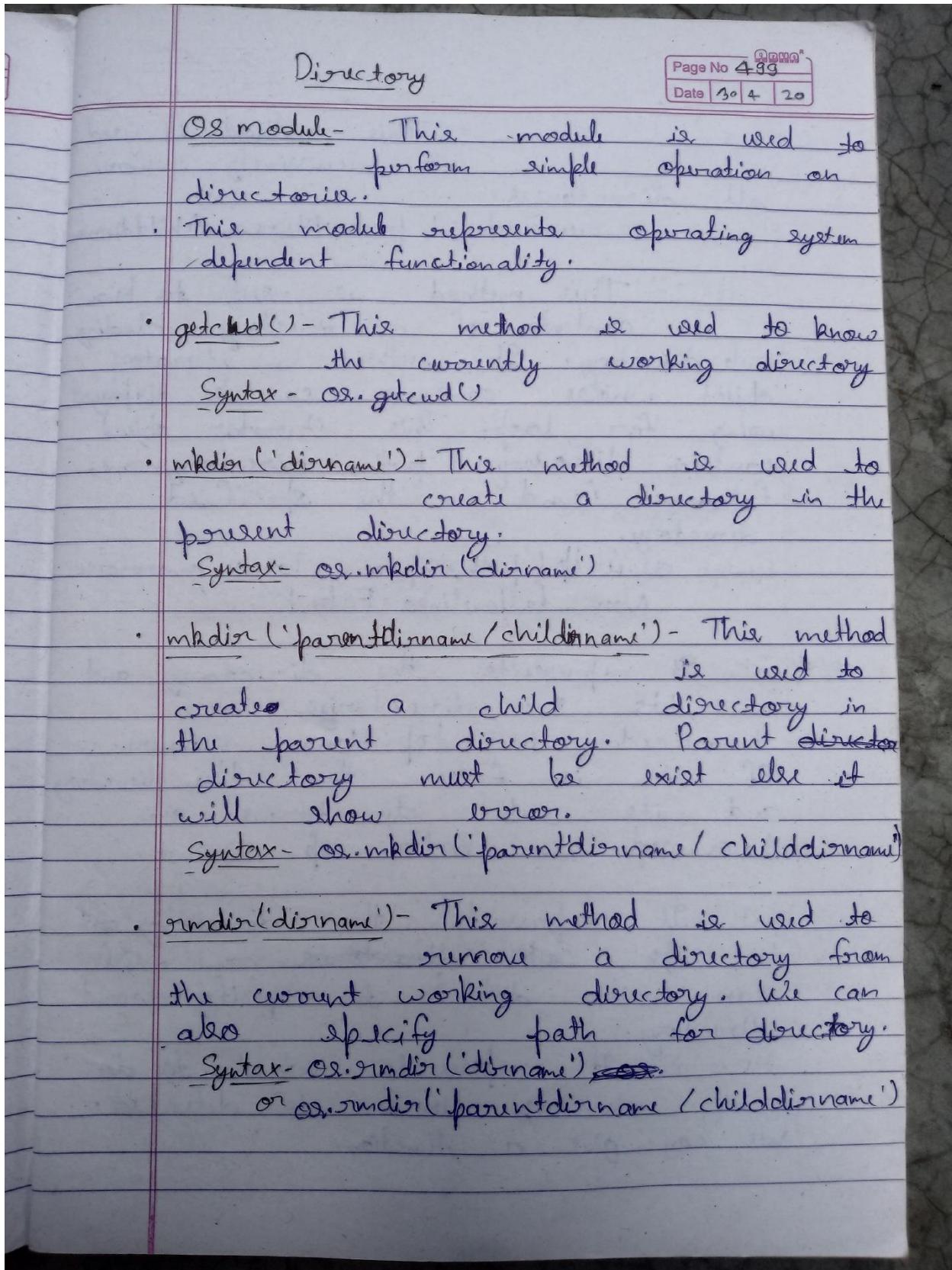
graph LR
    subgraph Pickling [Pickling]
        direction TB
        P1[stu1  
stu2] -- dump() --> P2[stu1  
stu2]
    end
    subgraph Unpickling [Unpickling]
        direction TB
        P2 -- load() --> U1[stu1]
    end

```

→ import pickle  
 class Student:  
 def \_\_init\_\_(self, name, roll, address):  
 self.name = name  
 self.roll = roll  
 self.address = address  
 def disp(self):  
 print(f'Name: {self.name} Roll: {self.roll}'  
 Address: {self.address})  
 with open('student.dat', 'wb') as f:  
 stu1 = Student('Satyam Seth', 101, 'Lucknow')  
 pickle.dump(stu1, f)  
 print('Pickling Done!!!')  
 with open('student.dat', 'rb') as f:  
 obj = pickle.load(f)  
 print('Unpickling Done!!!')  
 obj.disp()  
 Output - Pickling Done!!!







- Page No. 500  
Date \_\_\_\_\_
- removedirs('dirname') - This method is used to recursively remove all directories.  
Syntax: ex. `removedirs('parentdirname/childdirname')`
  - walk() - This method is used to know contents of a directory including sub directory. It returns an iterator object whose contents can be displayed using for loop. The iterator object contains directory path, directory name, filename found in the specified directory.  
Syntax: ex. `walk(path,topdown=True, onerror=None, followlinks=False)`
  - topdown - It represents the directory and its sub directories are traversed in top-down manner. If it is False then the directory and its sub directories are traversed in bottom-up manner.
  - path - It represents the directory and its sub directory name. We can write dot(.) to specify current directory.
  - onerror - It represents what to do when an error is detected. We can give a function.

Page No 501  
Date

• followlinks - True to visit directories pointed to by symbolic links, on systems that support them. If False walk() will not walk down into symbolic links that resolve to directories.

Note- os.makedirs() की सहायता से इस कोई रूप से folder नहीं बना सकता कि वह एक पारेंट के सिवाय किसी भी अन्य directory के नाम से बना हो।

FileExistError लाइब्रेरी / कि सहायता से इसका parent को os.makedirs() की सहायता से बना हो तो FileNotFoundError दिखाया जाता है। यदि वह एक ऐसा directory हो जो उसके parent directory (Parent) के नाम से बना हो तो FileNotFoundError दिखाया जाता है। इसका कारण यह है कि वह एक ऐसा directory है जो उसके parent को नहीं बना सकता है।

• makedirs('parentdir/childdir/grandchilddir') - This method is used to recursively create sub directories.

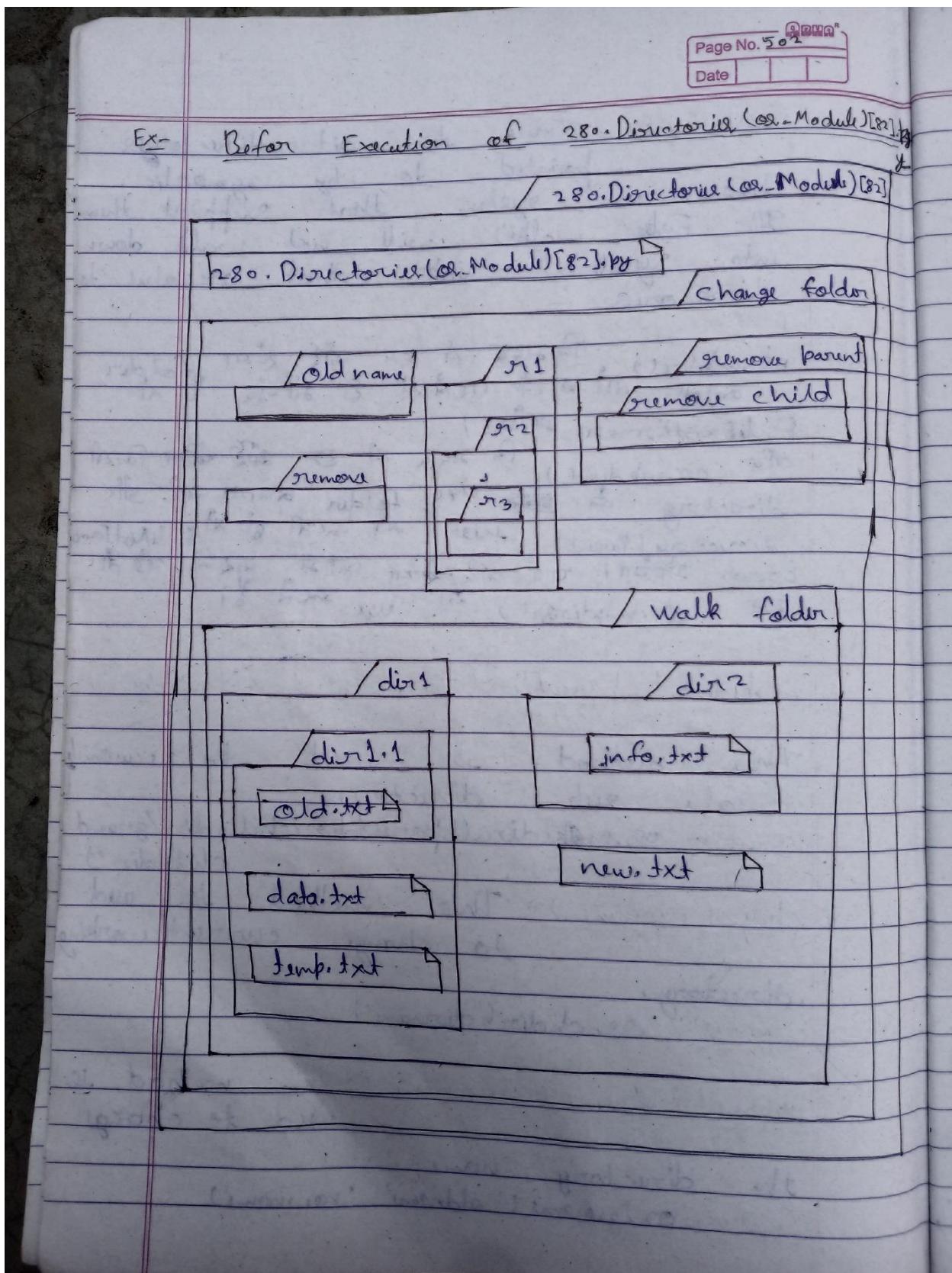
Syntax- os.makedirs('parentdir/childdir/grandchilddir')

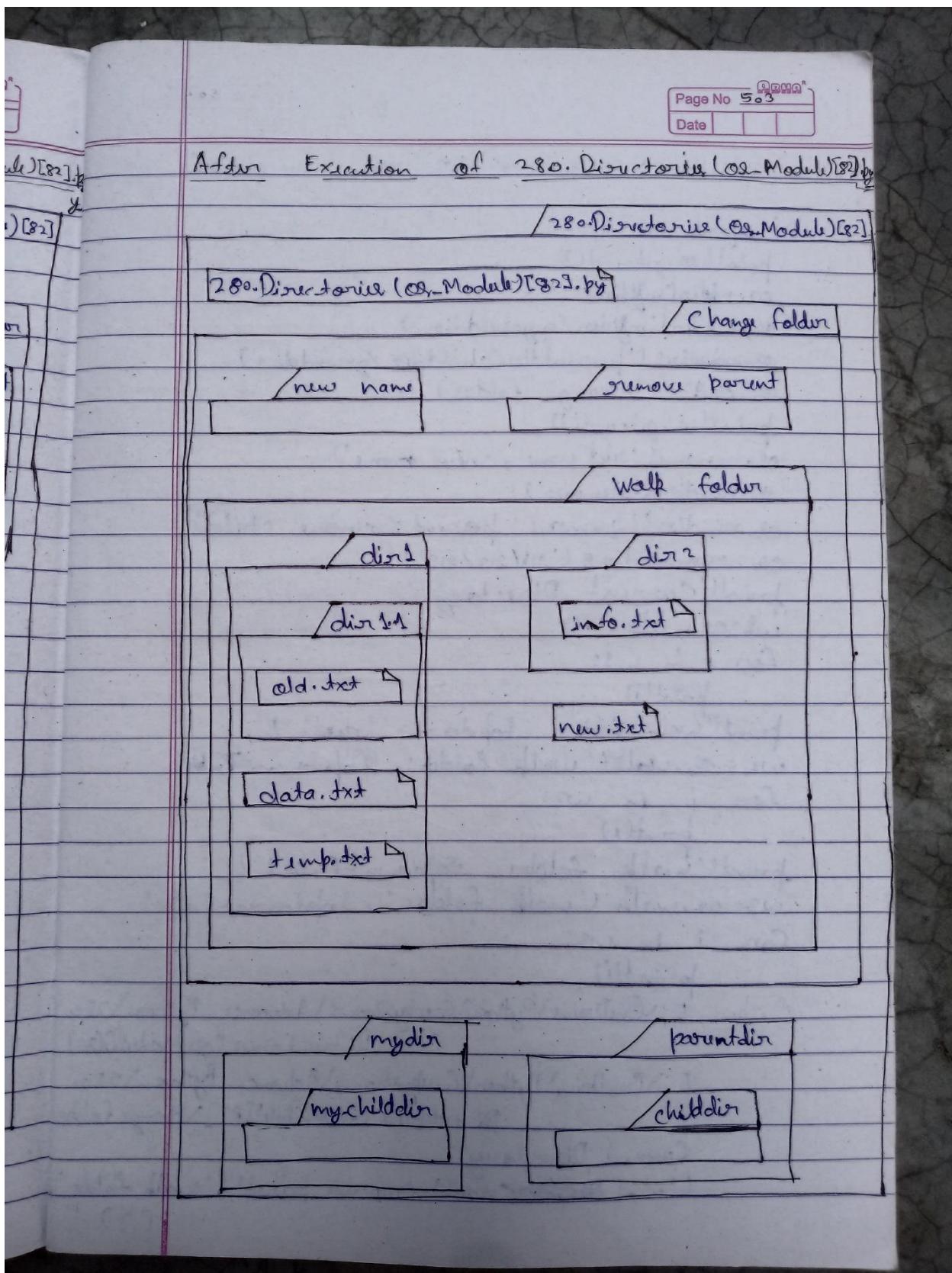
• chdir('dirname') - This method is used to change current working directory.

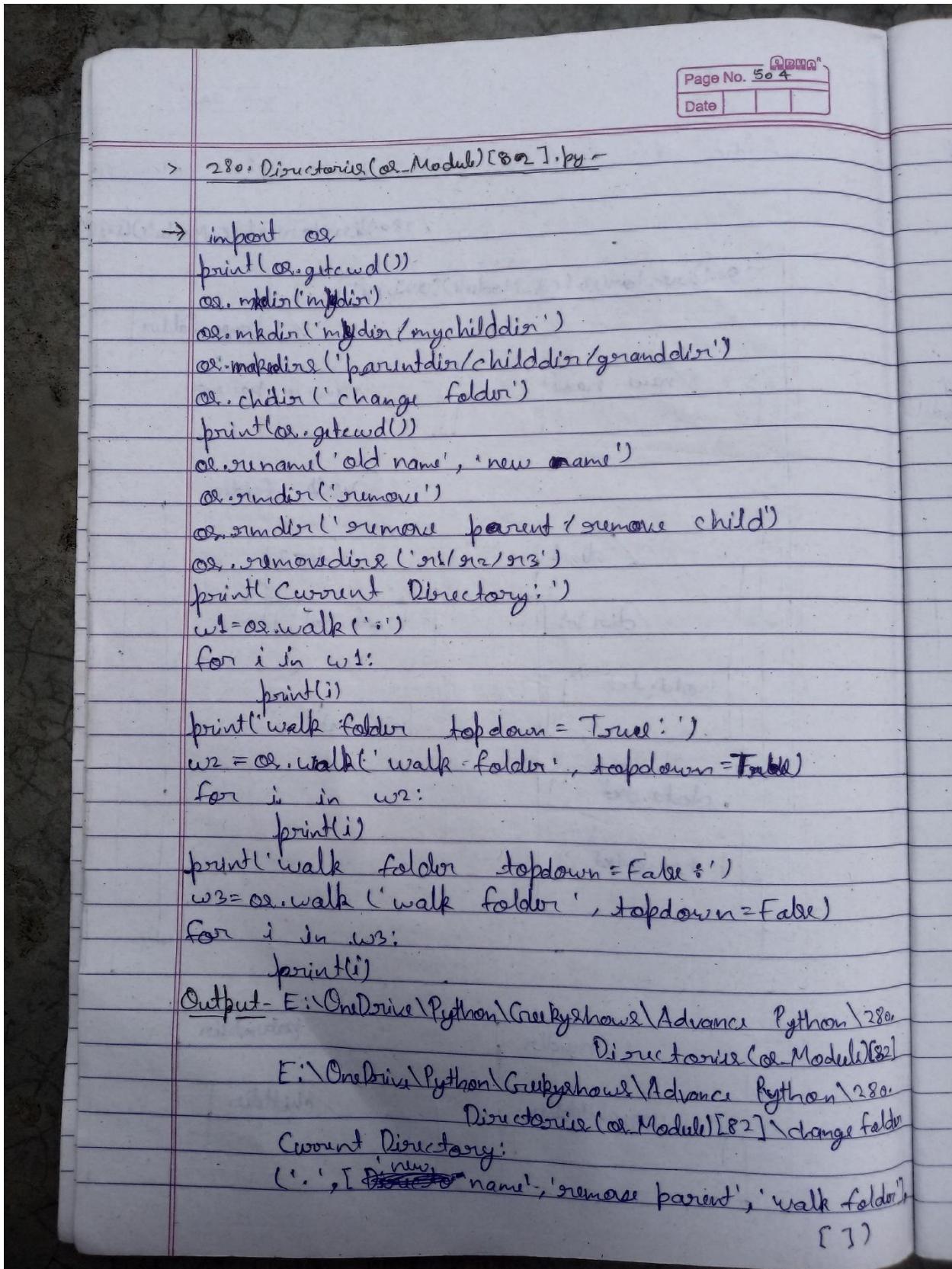
Syntax- os.chdir('dirname')

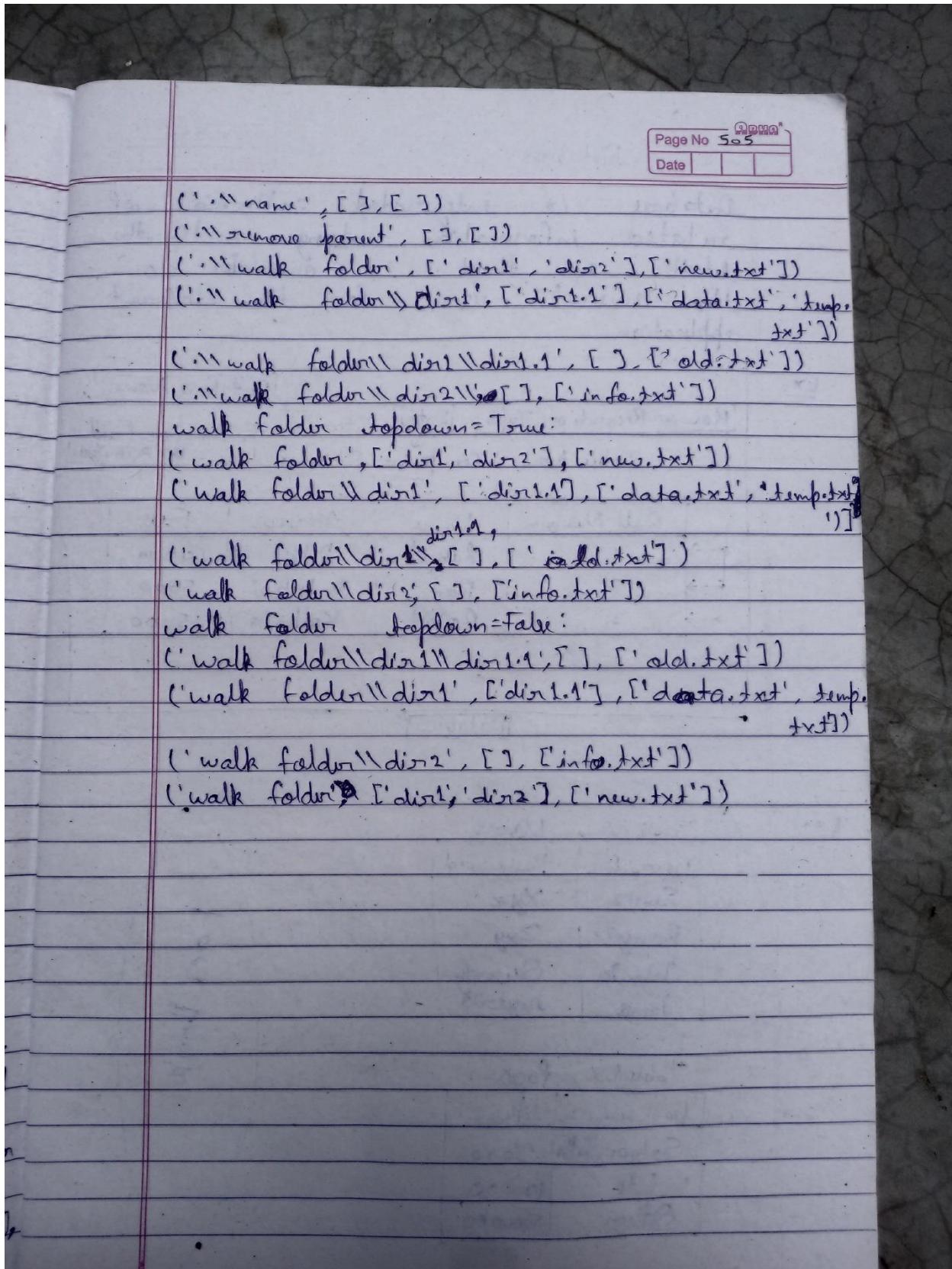
• rename('oldname', 'newname') - This method is used to change the directory name.

Syntax- os.rename('oldname', 'newname')









Database

Page No. 508  
Date

Database is integrated collection of related information along with the details so that it is available to the several user for the different application.

Ex-

Row or Record or Tuple      Entry      Student      Database Name

Table Name : Computer Science      Column or Field or Attribute

Roll Number	Name	Address	Fee
1	Rahul	Delhi	10000
2	Raj	Mumbai	5000
3	Rohit	Kolkata	15000

Database

Ex-

Table Name : User

user1id	PassWord
Sam12	Xyz
Rony23	Zxy
John20	Qwerty
Jame	Ingtz23

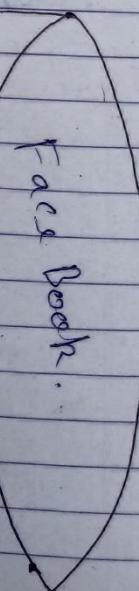
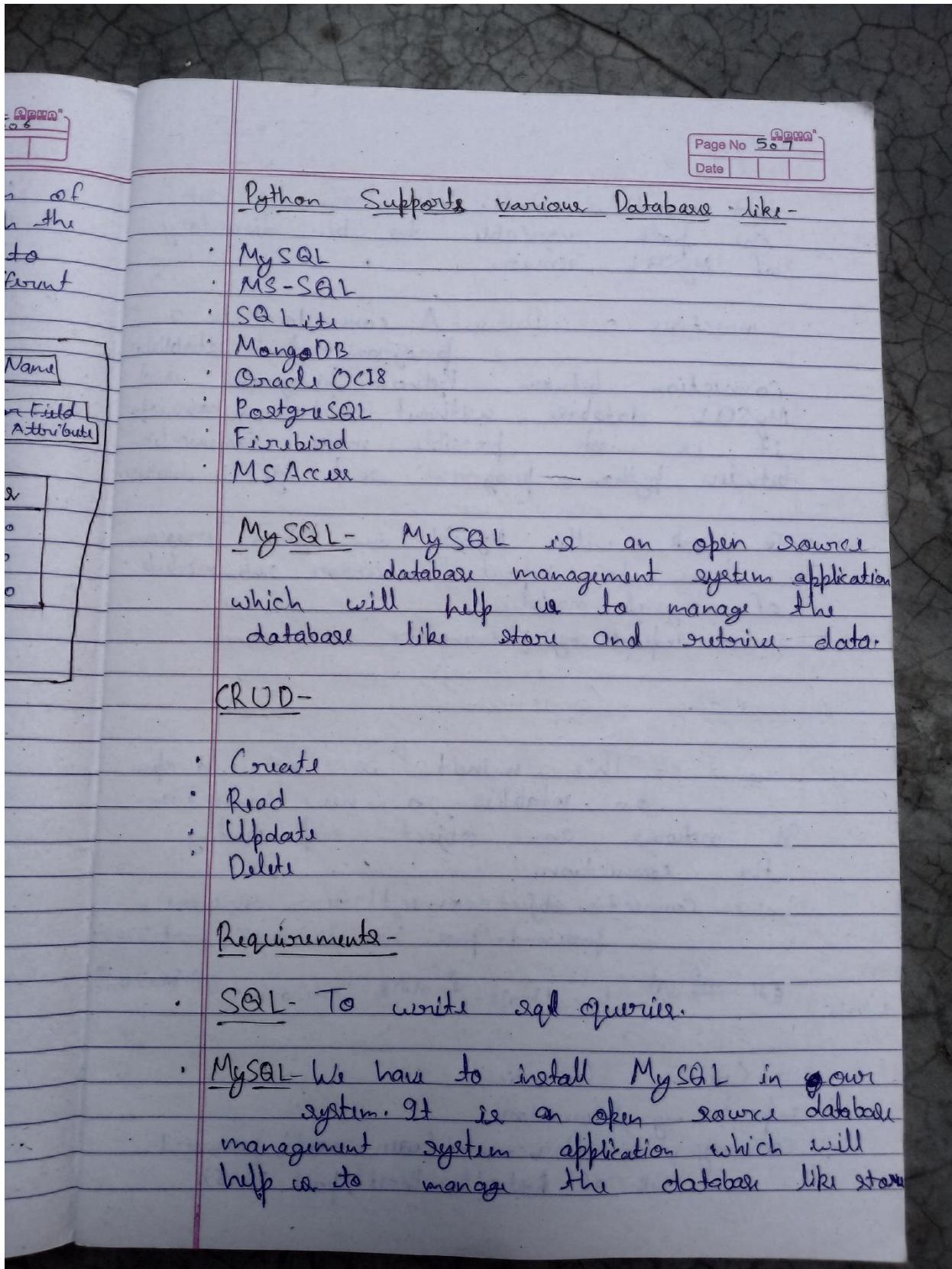
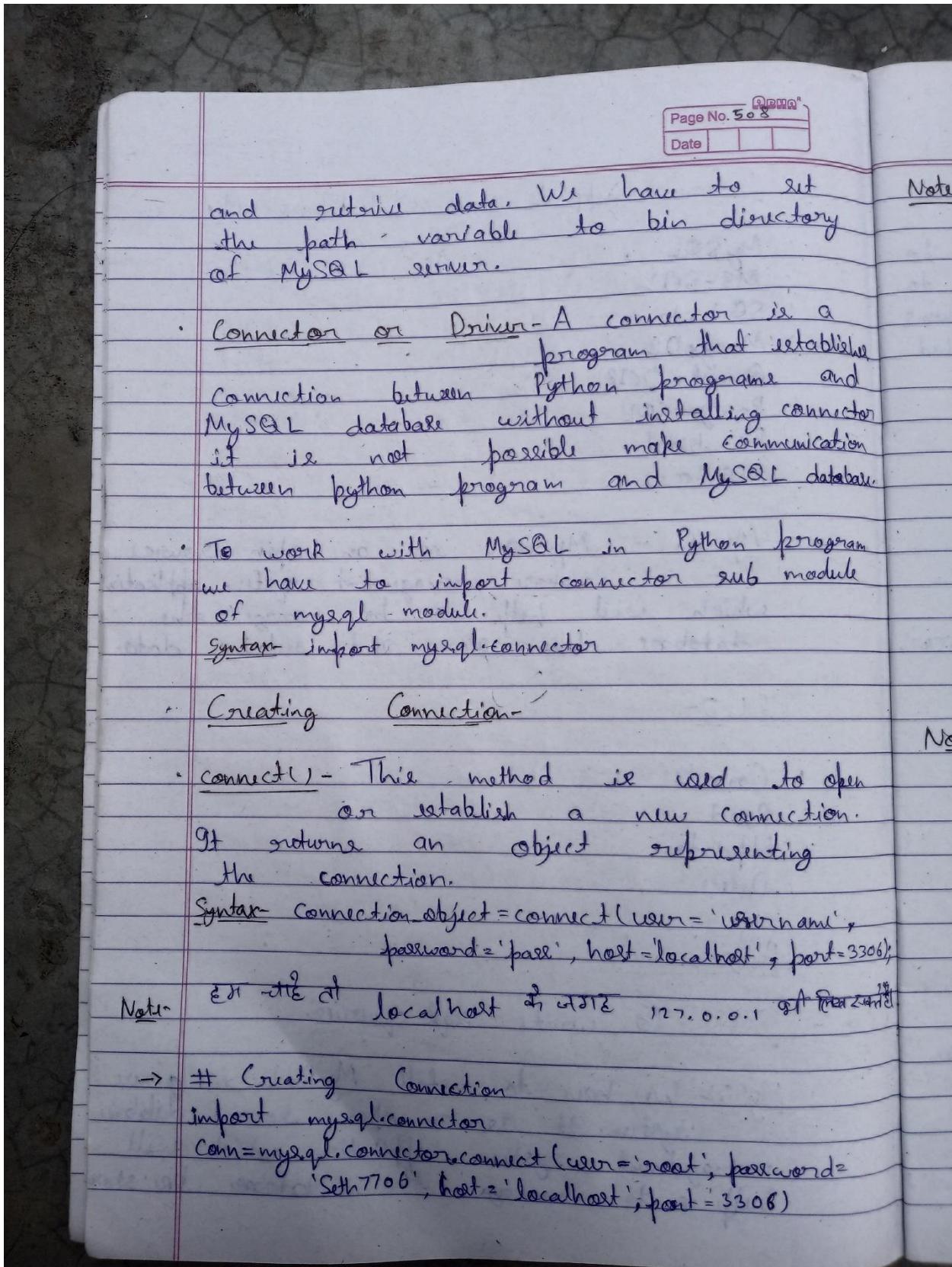


Table Name : Page

page_name	like
SatyamSeth	3000
Etc	100000
Other	5000000





Page No 509  
Date 15/05/20

set try Note- # Connection create करते वाले जैसी तरीकों  
जिनमें अप्पलॉड पासवर्ड का गलत हो जाए तो  
आधिक इसके बारे में exception handling का use  
करते हैं।

a tabulated and connection database. → # Creating Connection  
import mysql.connector  
try:  
    conn = mysql.connector.connect()  
    username = 'root',  
    password = 'Seth7707' # wrong password  
    host = 3306,  
    host = '127.0.0.1'  
except:  
    print('Unable To Connect!')  
Output- Unable To Connect!

open on. Note- # It is at first arguments की dictionary तरीके  
connect() के \*\*kwargs पर जो भी भीड़ हो सकती है।

→ # Creating Connection  
import mysql.connector  
config = {  
    'username': 'root',  
    'password': 'Seth7707',  
    'host': 'localhost',  
    'port': 3306  
}  
try: ~~except:~~  
    conn = mysql.connector.connect(\*\*config)  
except:  
    print('Unable To Connect!')

Page No. 510

Check Connection -

- is\_connected() - This method is used to check if the connection to MySQL is established or not. It returns True if the connection is established successfully.

Syntax - connection\_object.is\_connected()

```

→ # Creating Connection
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706', or password='Seth7707',
        host='localhost',
        port=3306
    )
    if(conn.is_connected()):
        print('Connected')
    except:
        print('Unable To Connect')

```

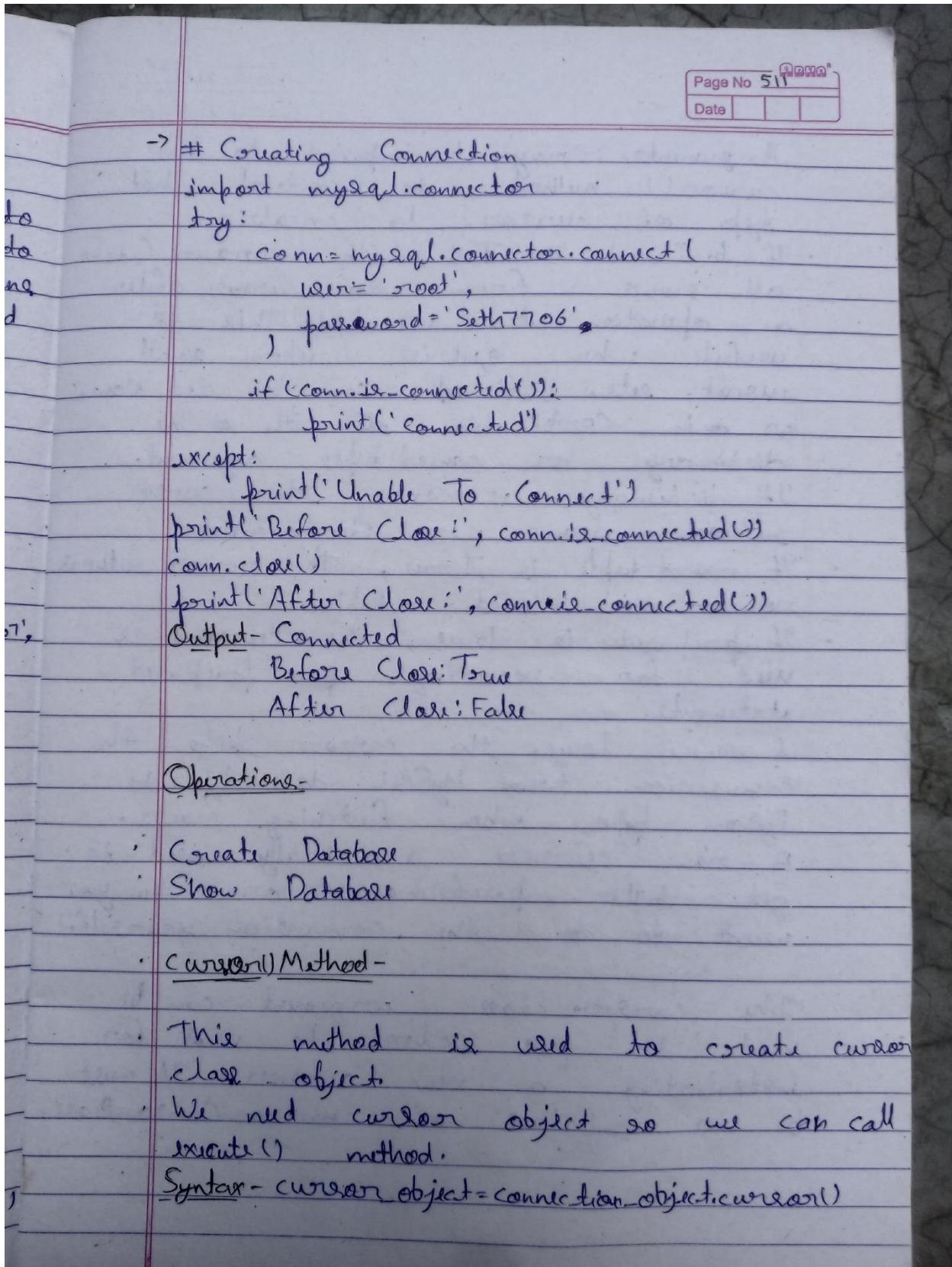
Output - Connected or Unable To Connect

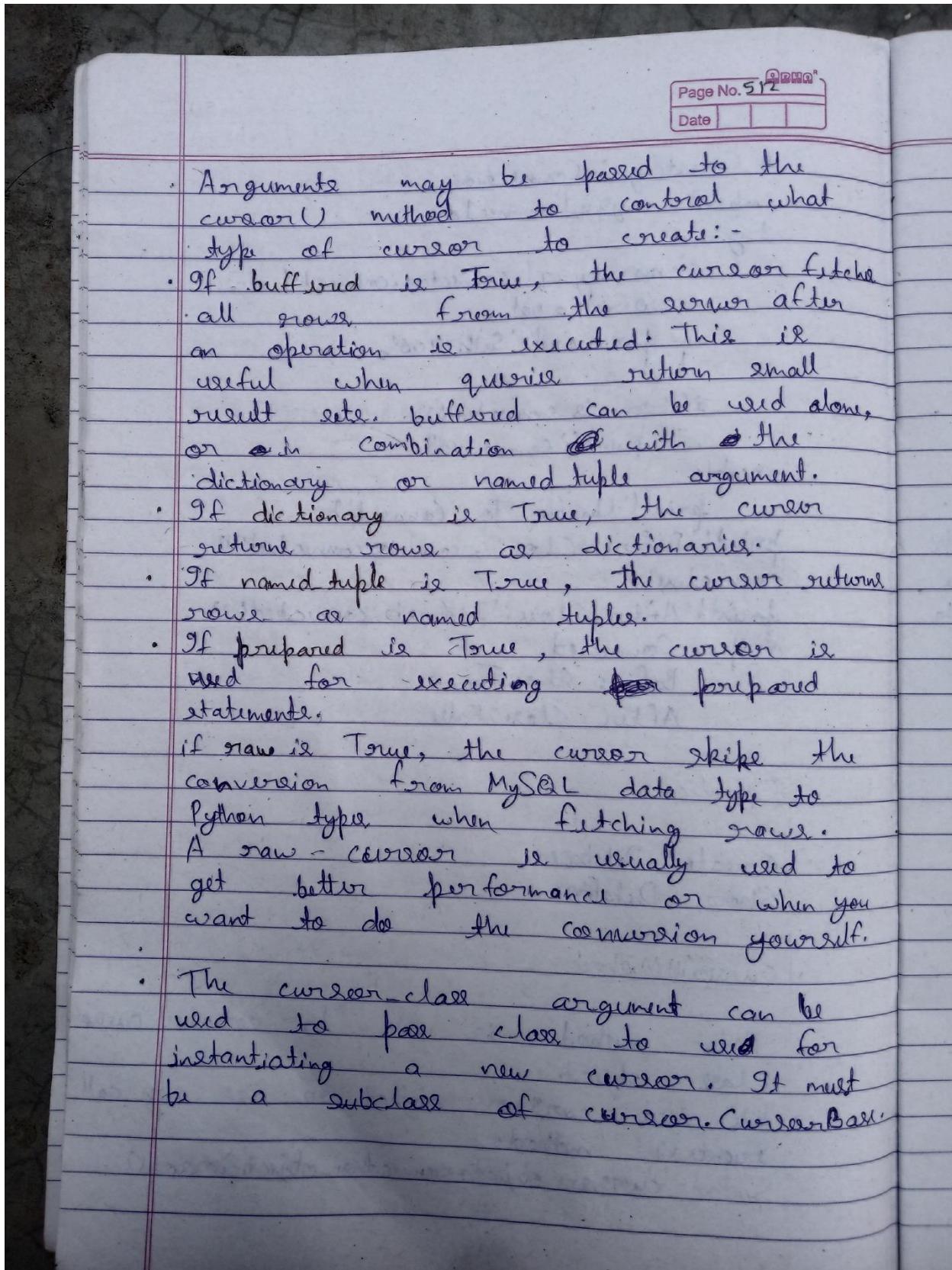
Close Connection -

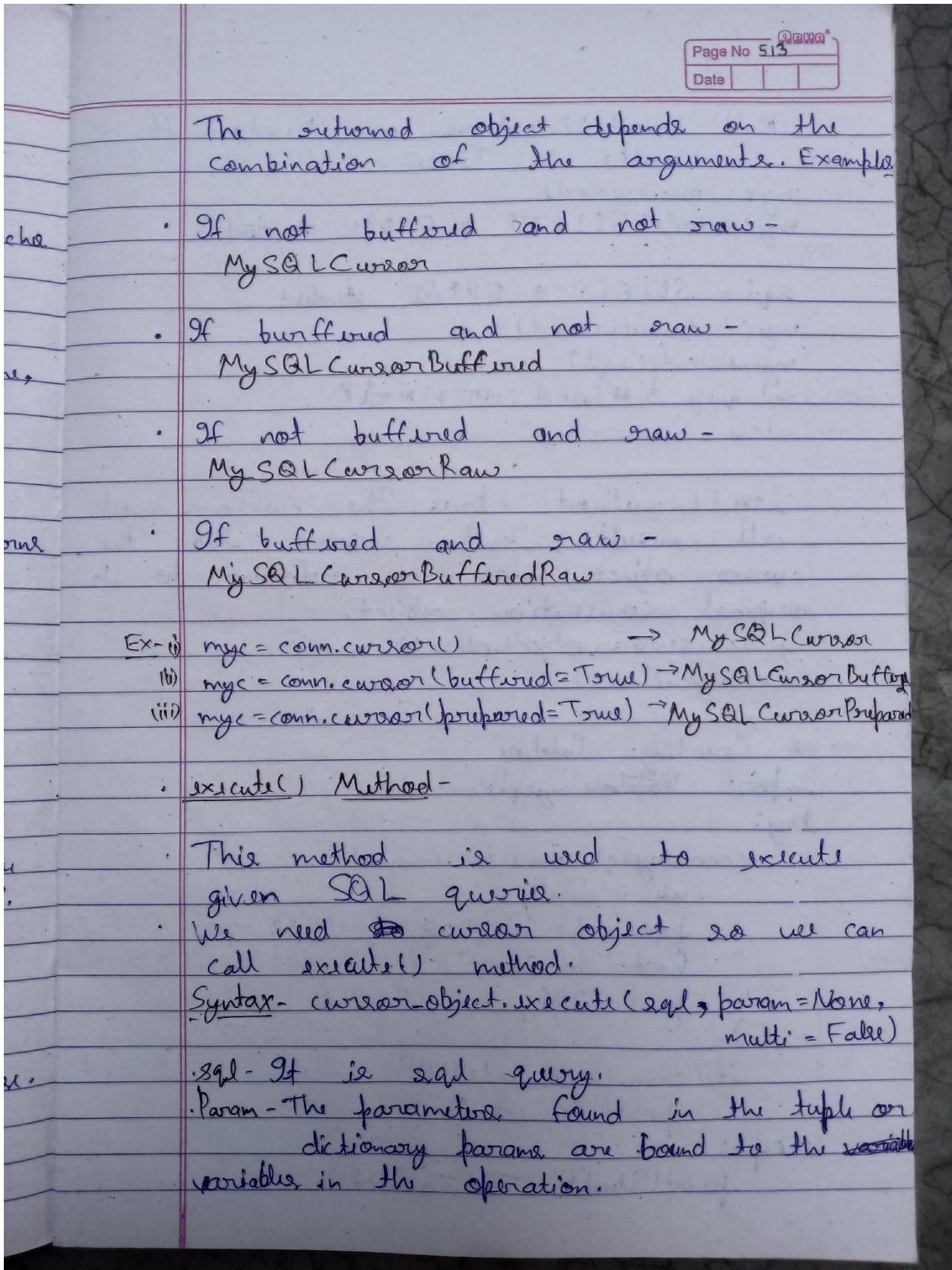
- close() - This method is used to close the connection.

Syntax - Connection\_object.close()

Note - ~~जटिल नहीं है।~~ Port 3306 host default एवं जटिल है।







APUNTS  
Page No. 514  
Date \_\_\_\_\_

• `Multi_execut()` returns an iterator if `multi` is True.

Ex-  
`myc = conn.cursor()  
myc.execute('SELECT * FROM student')`

on  
`sql = 'SELECT * FROM student'  
myc = conn.cursor()  
myc.execute(sql)`  
Note- SQL query & last ; must be used

Close Cursor -

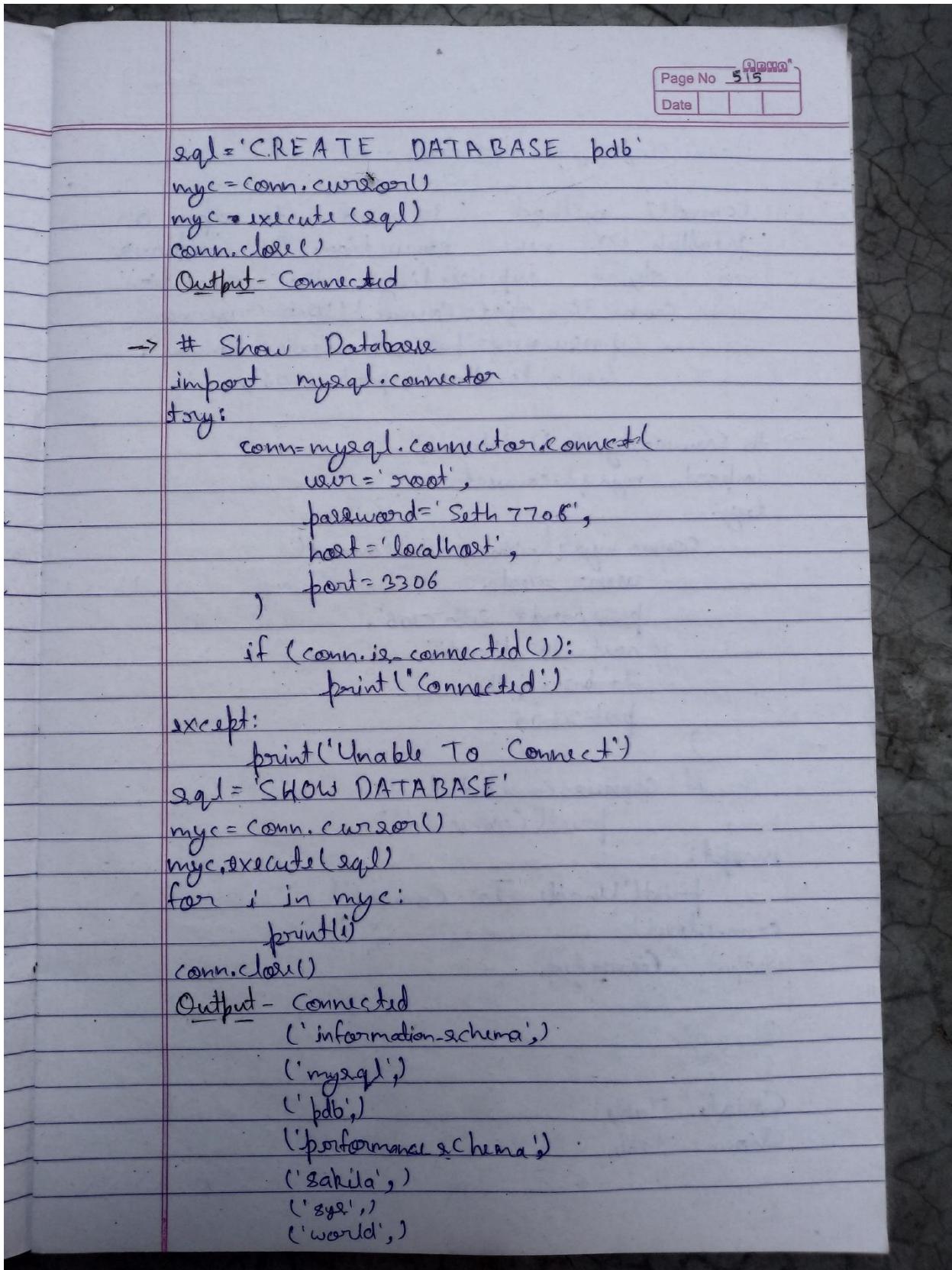
`close()` method closes the cursor, write all results, and ensure that the cursor object has no reference to its original connection object.

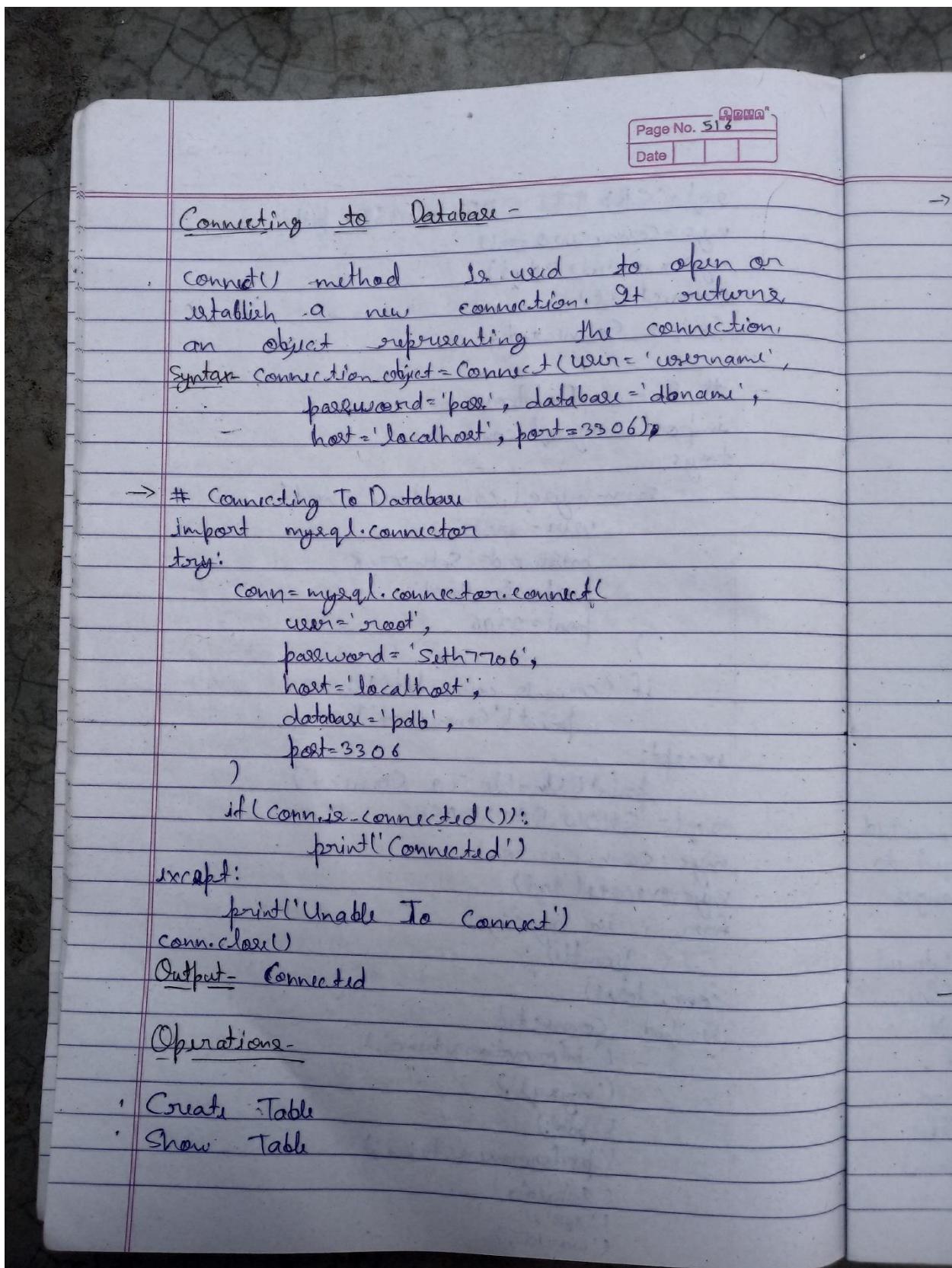
Syntax- `cursor_object.close()`

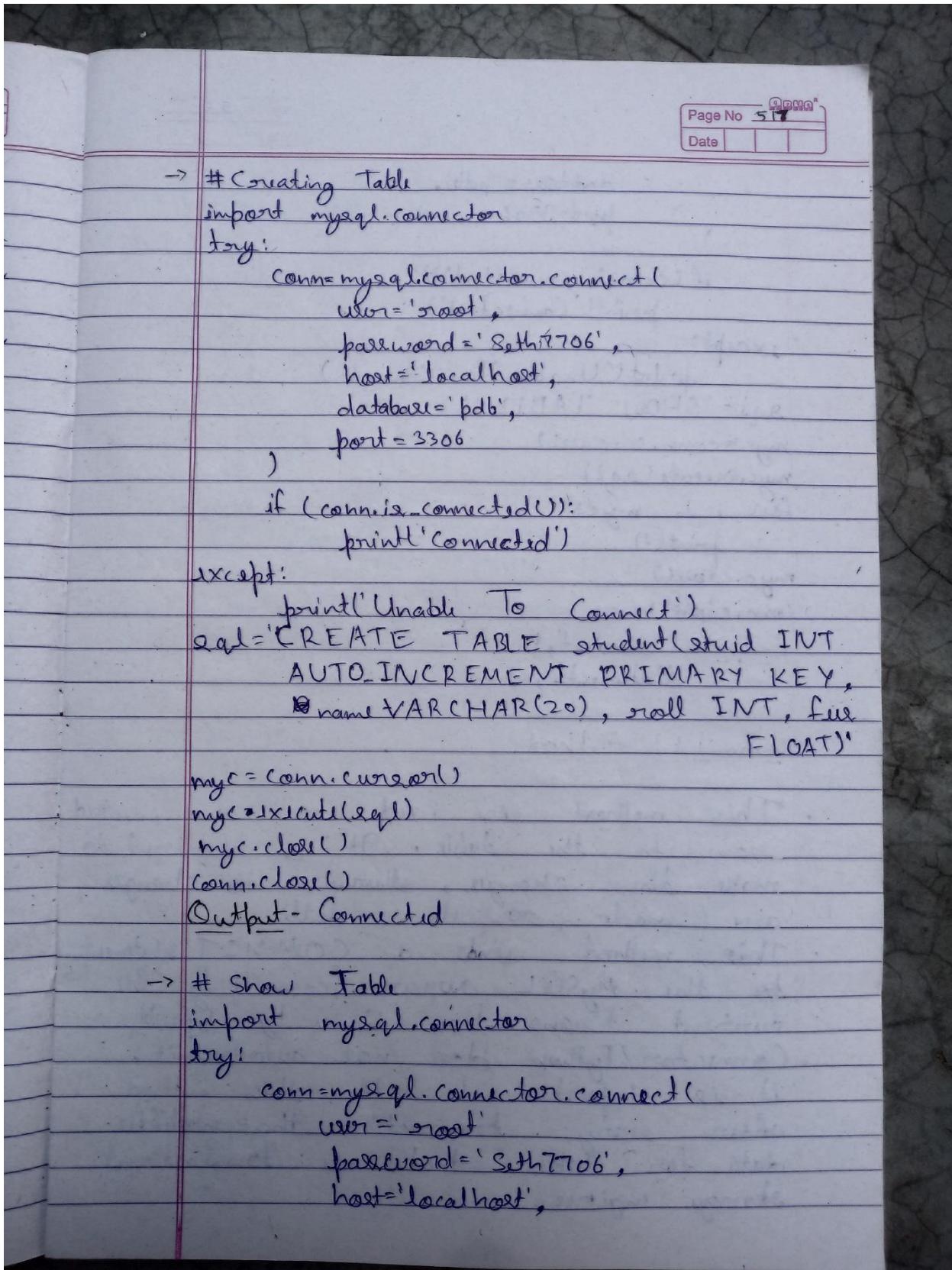
Ex- `myc.close()`

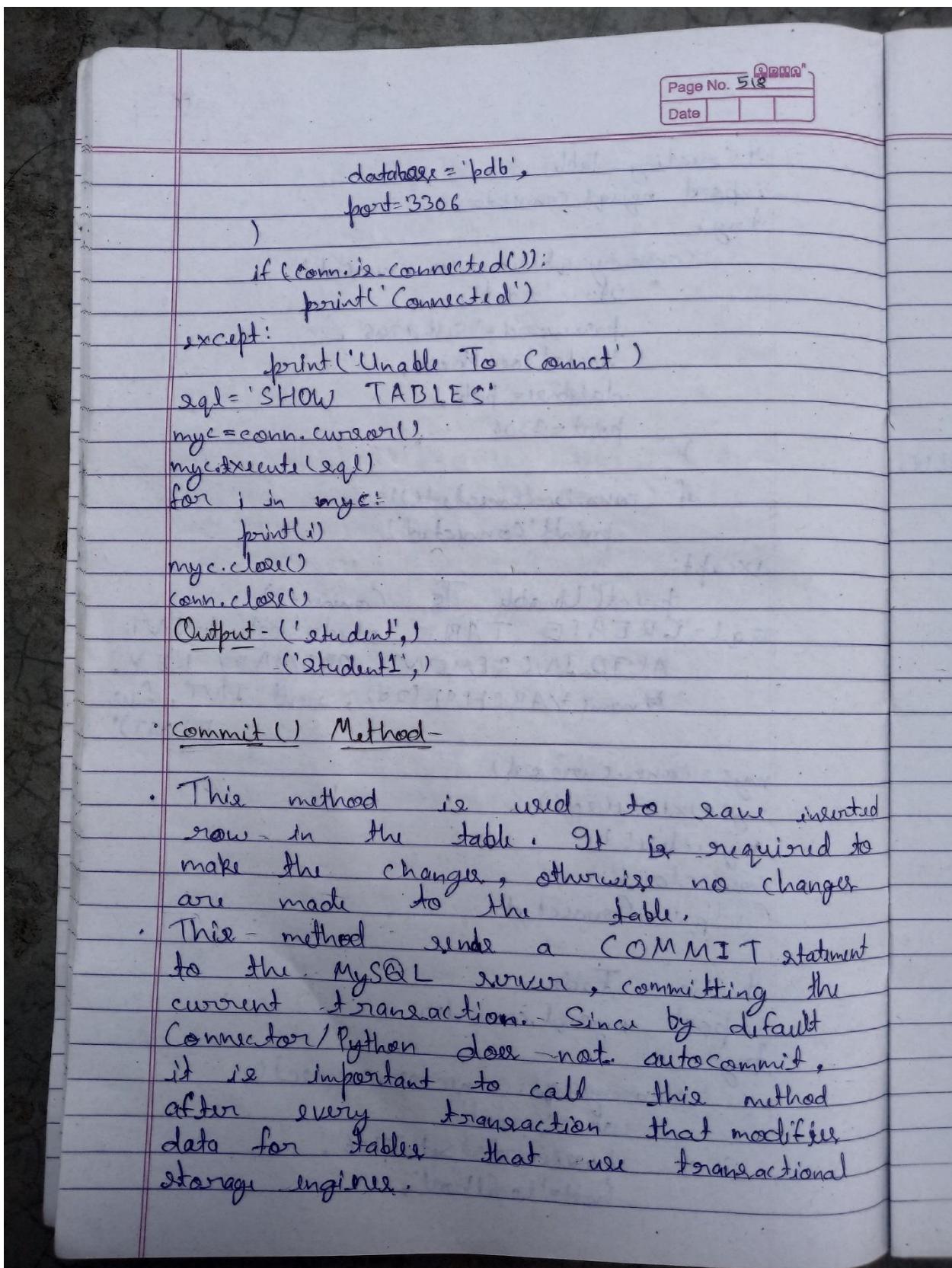
→ # Creating Database

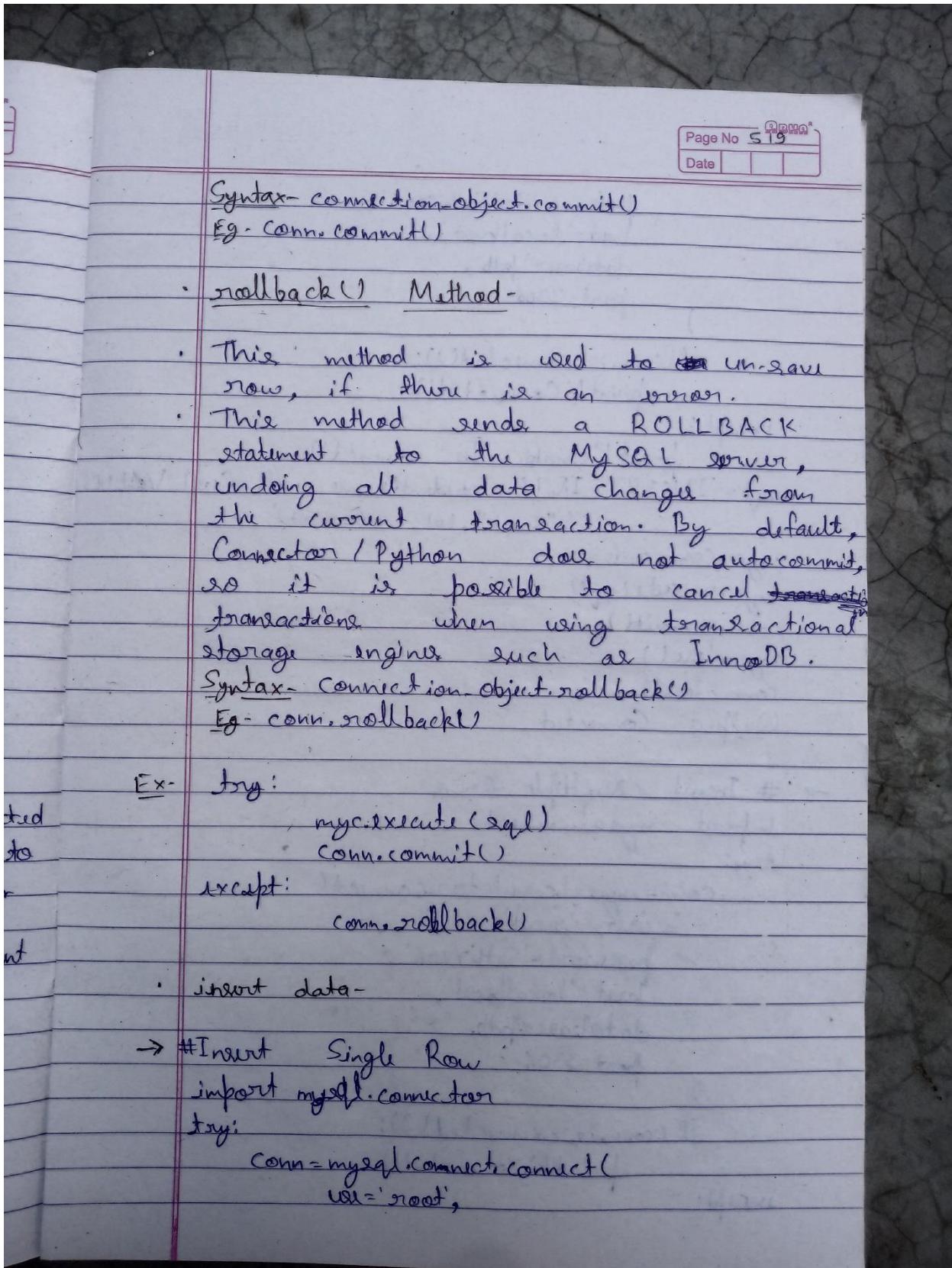
```
import Database mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
```











Page No. 520  
Date

```

password='Seth7706',
host='localhost',
database='pdb',
port=3306
)
if (conn.is_connected()):
    print('Connected')
except:
    print('Unable To Connect')
sql='INSERT INTO student(name, roll, fee) VALUES
    ("Sumit", 101, 50600.52)'
myc=conn.cursor()
myc.execute(sql)
conn.commit()
myc.close()
conn.close()
Output - Connected

→ # Insert Multiple Rows
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:

```

Page No 521  
Date

```

print('Unable To Connect')
sql='INSERT INTO student(name, roll, fee)
VALUES ("Jai", 102, 45843.5), ("Kuru", 103,
45201.7), ("Baaanti", 104, 48451.80)
myc = conn.cursor()
try:
    myc.execute(sql)
    conn.commit()
    print('Rowe Inserted')
except:
    conn.rollback()
    print('Unable To Insert Data')
myc.close()
conn.close()

Output- Connected
Rowe Inserted

```

rowcount Property - This read-only property returning the number of rows returned for SELECT statements, or the number of rows affected by DML statements such as INSERT or UPDATE.

Syntax - cursorobject.rowcount

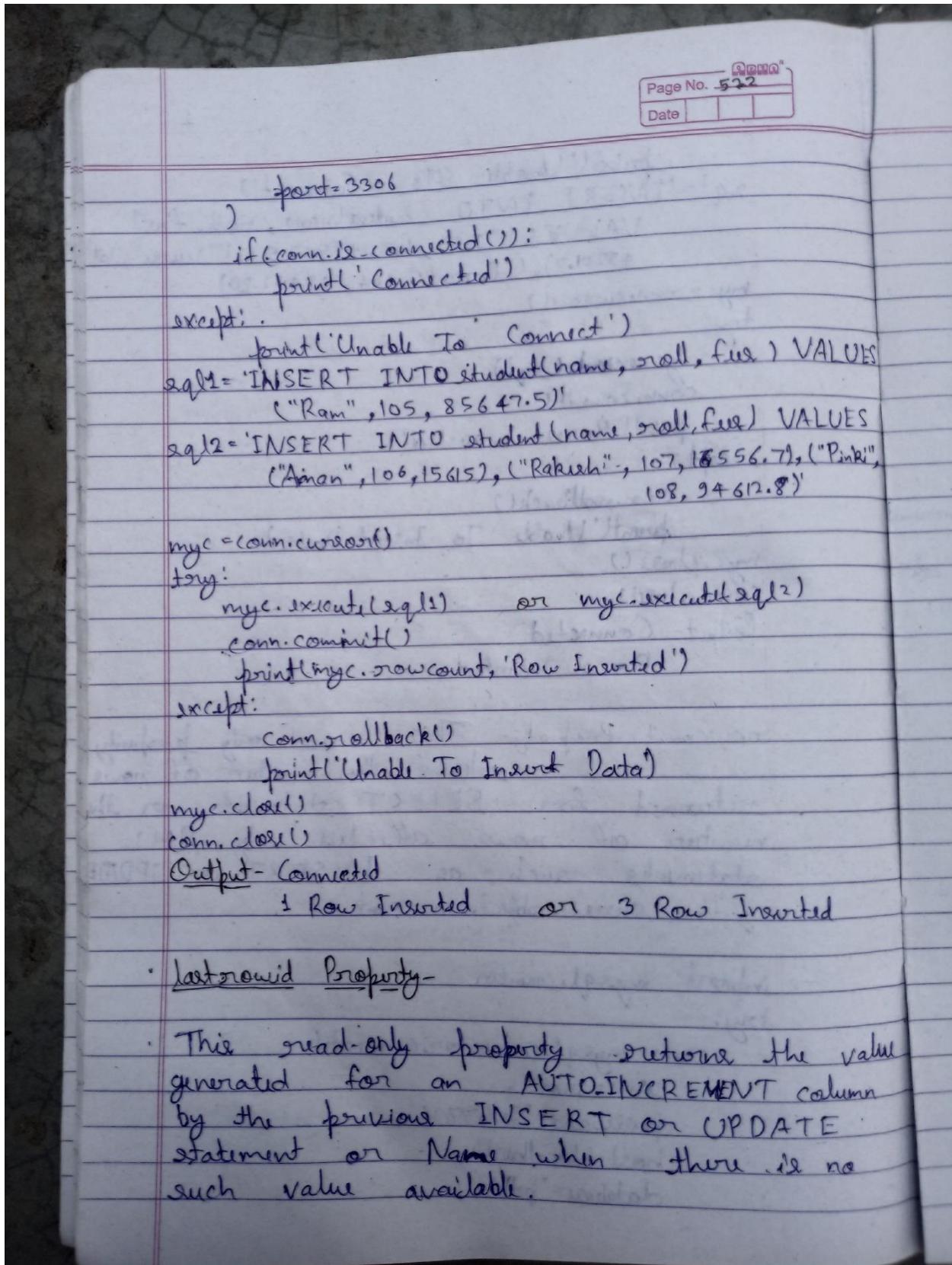
→ import mysql.connector

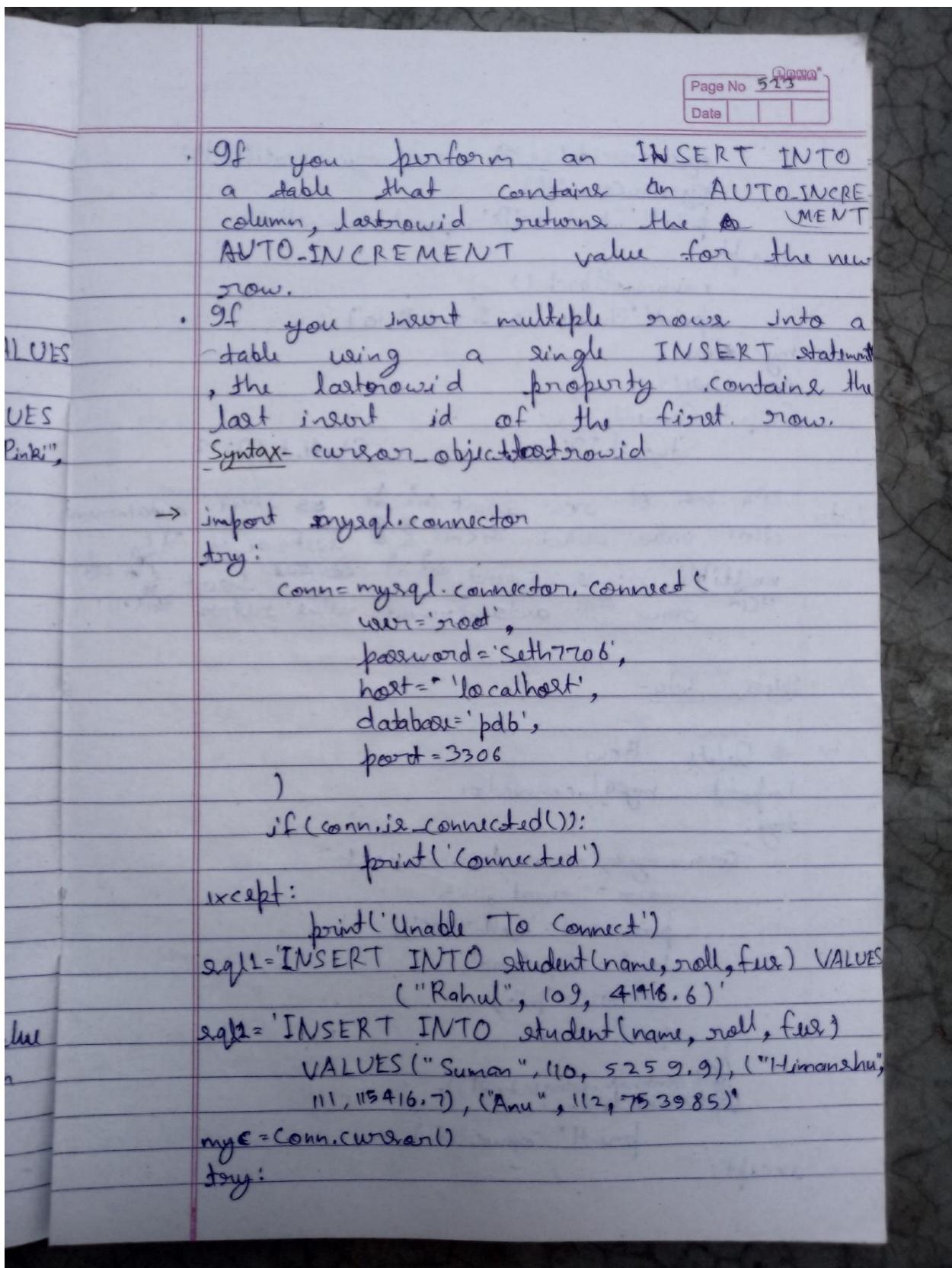
try:

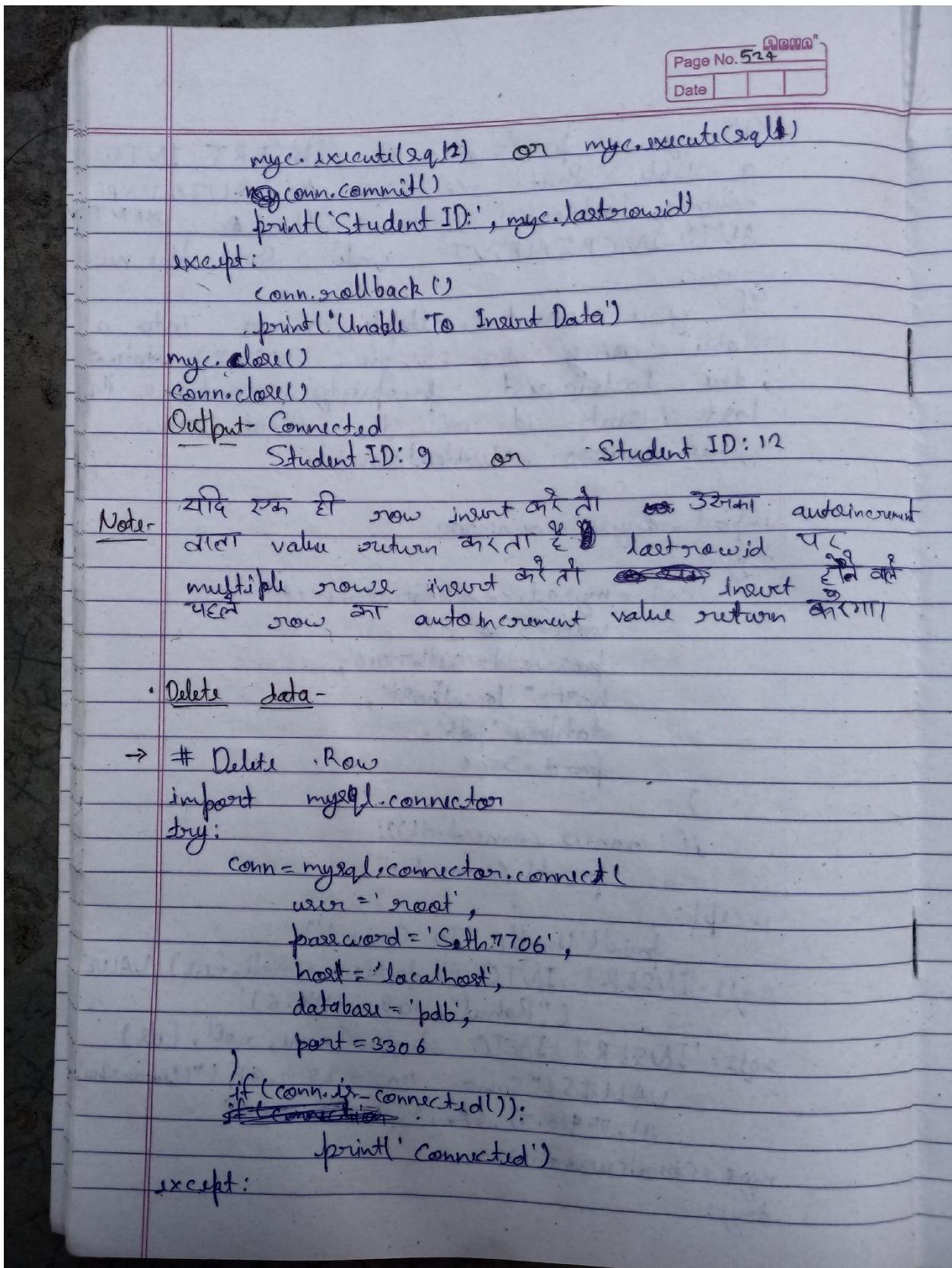
```

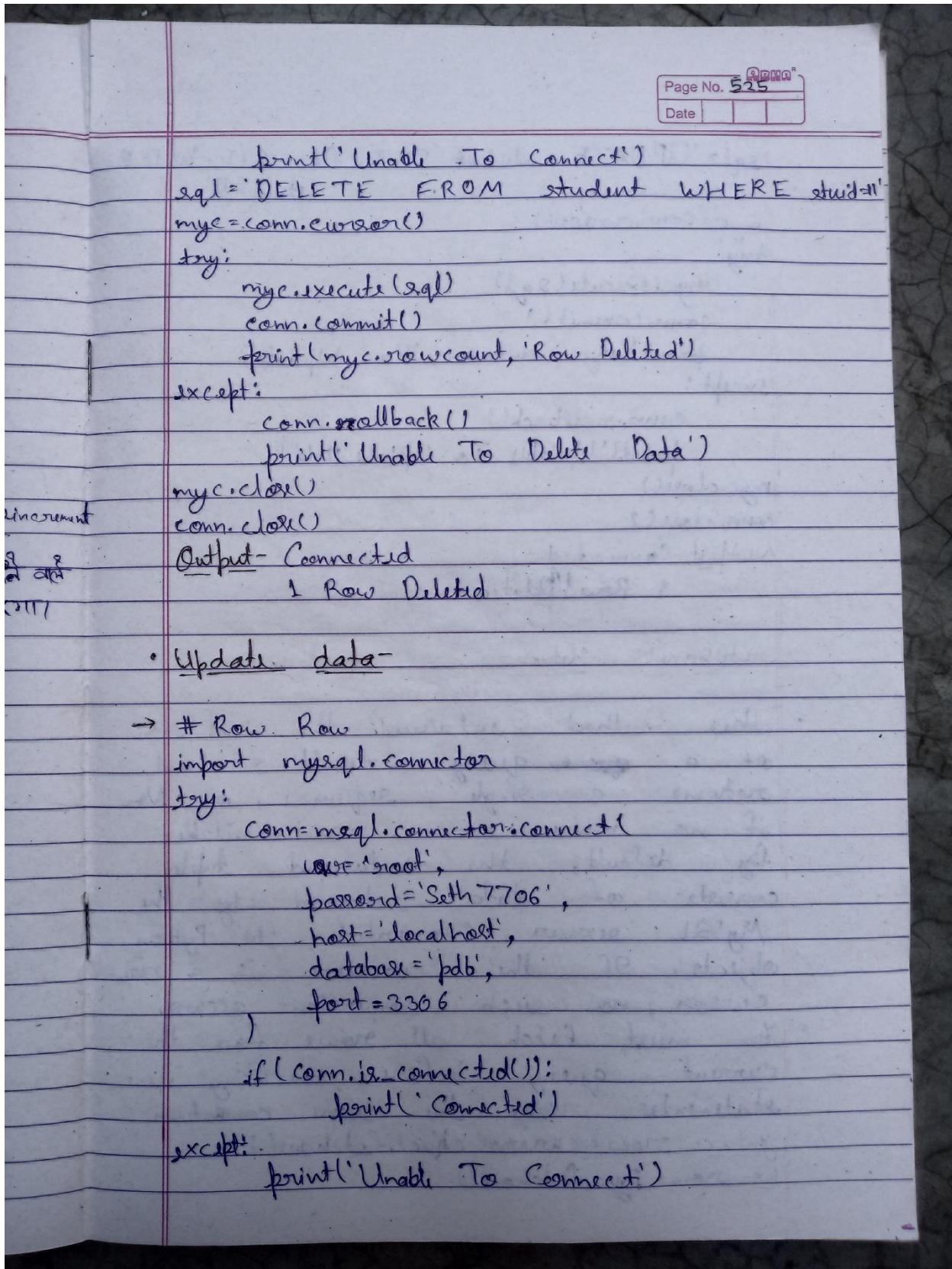
conn = mysql.connector.connect(
    user='root',
    password='Seth7706',
    host='localhost',
    database='polki',
)

```









Page No. 526  
Date

`sql = 'UPDATE student SET fee = 625 WHERE stuid = 5'` →

`myc = conn.cursor()`

`try:`

`myc.execute(sql)`

`conn.commit()`

`print(myc.rowcount, "Row Updated")`

`except:`

`conn.rollback()`

`print("Unable To Update Data")`

`myc.close()`

`conn.close()`

Output - Connected

1 Row Updated.

### fetchone() Method -

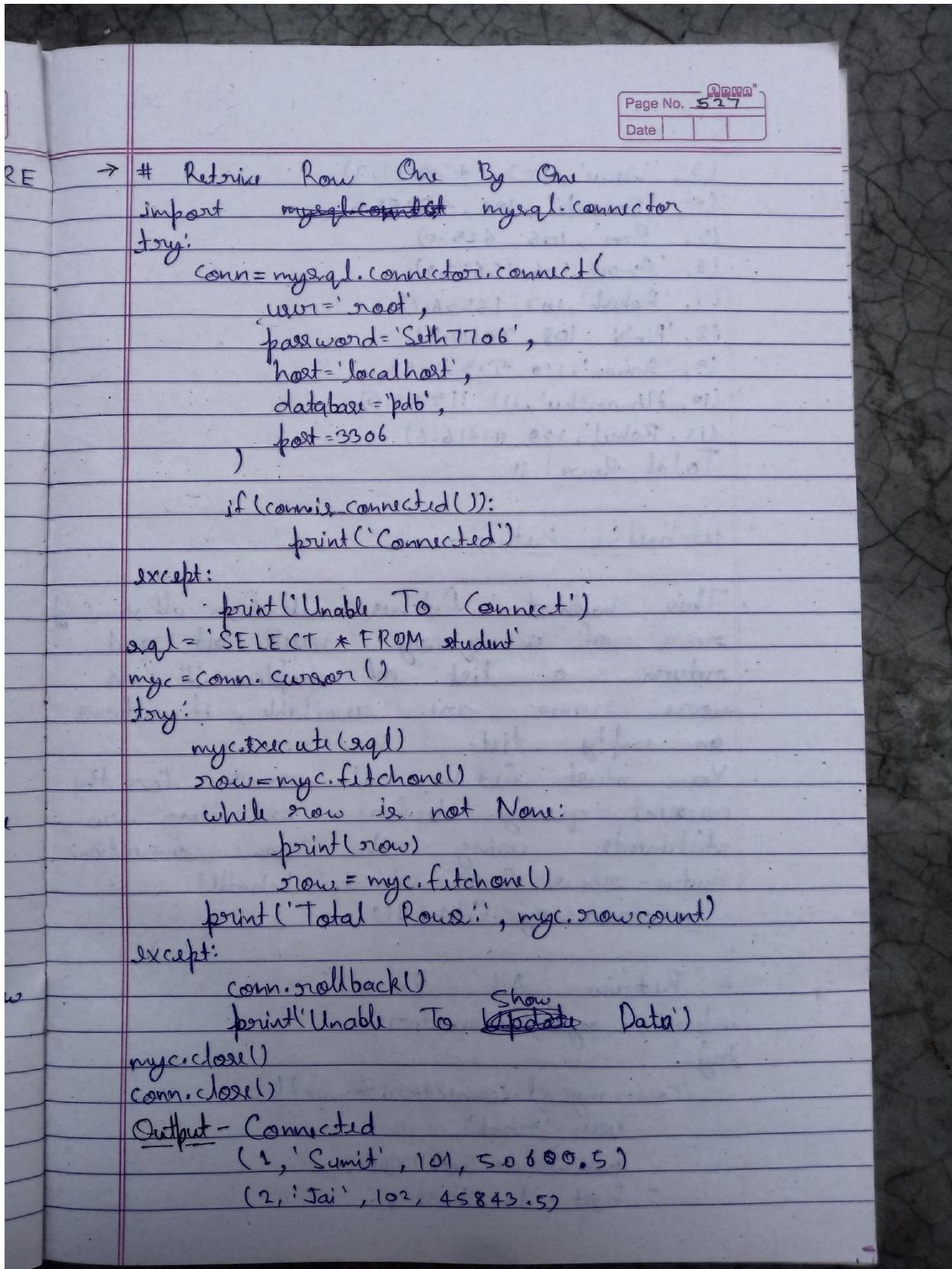
- This method retrieves the next row of a ~~query~~ query result set and returns a single sequence, or None if no more rows are available.

By default, the returned tuple consists of data returned by the MySQL server, converted to Python objects. If the cursor is a raw cursor, no such conversion occurs.

You must fetch all rows for the current query before executing new statements using the same connection.

Syntax - `row = cursor_object.fetchone()`

Ex - `row = myc.fetchone()`



Page No. 528  
Date

(3, 'Venu', 103, 45201.7)
(4, 'Basanti', 104, 48451.8)
(5, 'Ram', 105, 625.0)
(6, 'Aman', 106, 15615.0)
(7, 'Rakesh', 107, 16556.7)
(8, 'Pinki', 108, 94612.8)
(9, 'Suman', 110, 52595.9)
(10, 'Himanshu', 111, 115417.0)
(12, 'Rahul', 109, 41416.6)

Total Rows: 11

• fatchall() Method -

- This method fetches all (or all remaining) rows of a query result set and returns a list of tuple. If no more rows are available, it returns an empty list.
- You must fetch all rows for the current query before executing new statements using the same connection.

Syntax - `rows = CursorObject.fetchall()`

Ex. `rows = myc.fetchall()`

```

→ # Retrieve All Rows
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
    )

```

Page No. 529 ARNA Date

```

        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
    except:
        print('Unable To Connect')
    sql='SELECT * FROM student'
    myc=conn.cursor()
    try:
        myc.execute(sql)
        rows=myc.fetchall()
        for row in rows:
            print(row)
        print('Total Rows:', myc.rowcount)
    except:
        conn.rollback()
        print('Unable To Show Data')
    myc.close()
    conn.close()

```

Output - Connected

(1, 'Sumit', 101, 50600.5)
(2, 'Jai', 102, 45843.5)
(3, 'Vernu', 103, 45201.7)
(4, 'Basanti', 104, 48451.8)
(5, 'Rani', 105, 625.0)
(6, 'Aman', 106, 15615.0)
(7, 'Rakesh', 107, 16556.7)
(8, 'Pinki', 108, 94612.8)
(9, 'Suman', 110, 52595.9)
(10, 'Mimaneshu', 111, 115417.0)

Page No. 53  
Date

(12, 'Rahul', 'Seth', 109, 41216, 6)  
Total Rows: 11

- fetchmany() Method -
- This method fetches the next set of rows of ~~so~~ a query result and returns a list of tuples. If no more rows are available, it returns an empty list.
- The number of rows returned can be specified using the size argument, which defaults to one. Fewer rows are returned if fewer rows are available than specified.
- You must fetch all rows for the current query before executing new statements using the same connection.

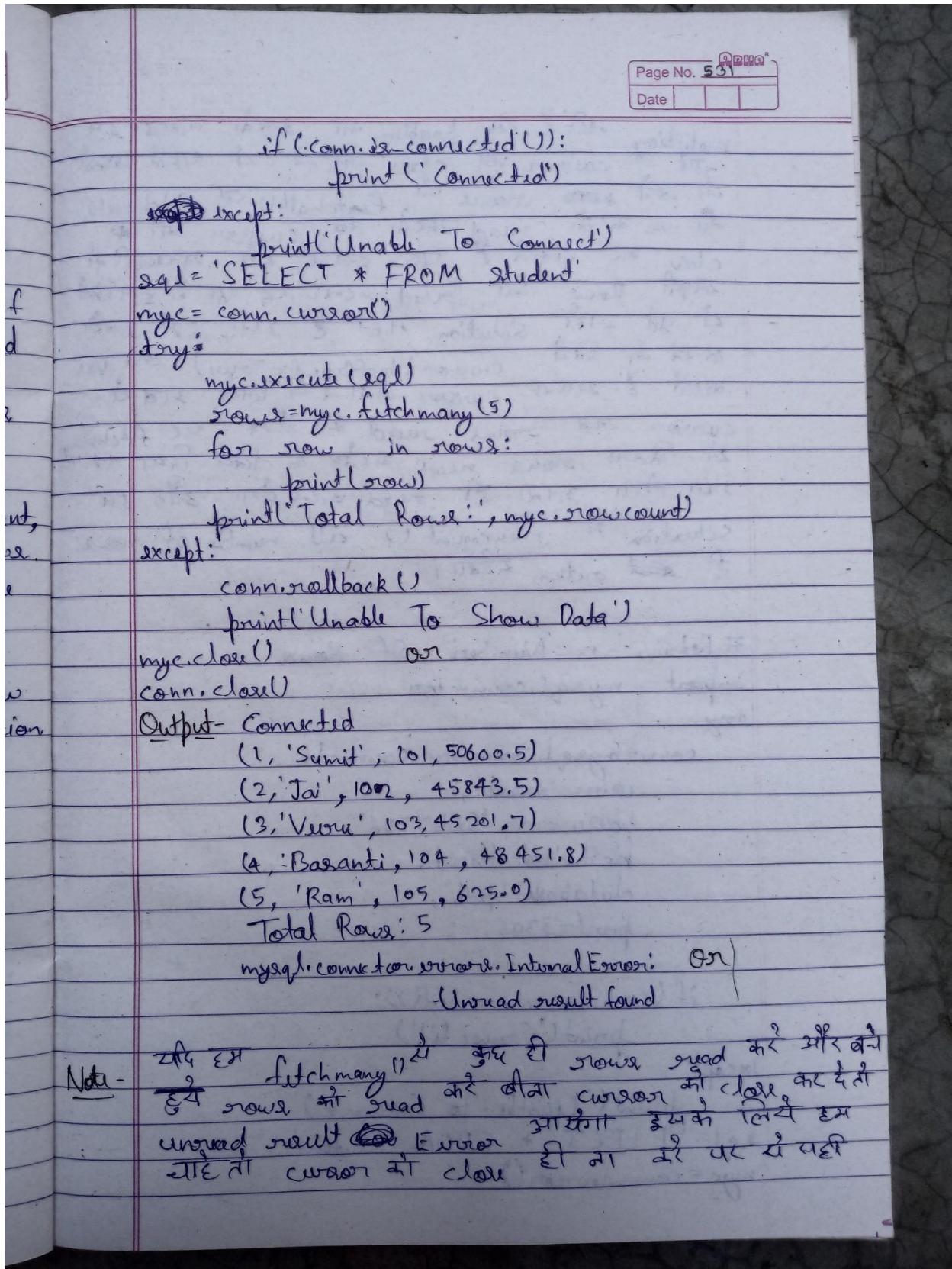
Syntax - rows = cursor\_object.fetchmany(size=1)

Ex - rows = myc.fetchmany(3)

→ #Retrive n Number Of Rows

```
import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    print("Connection successful")
except Exception as e:
    print(f"Error: {e}")
```

Note



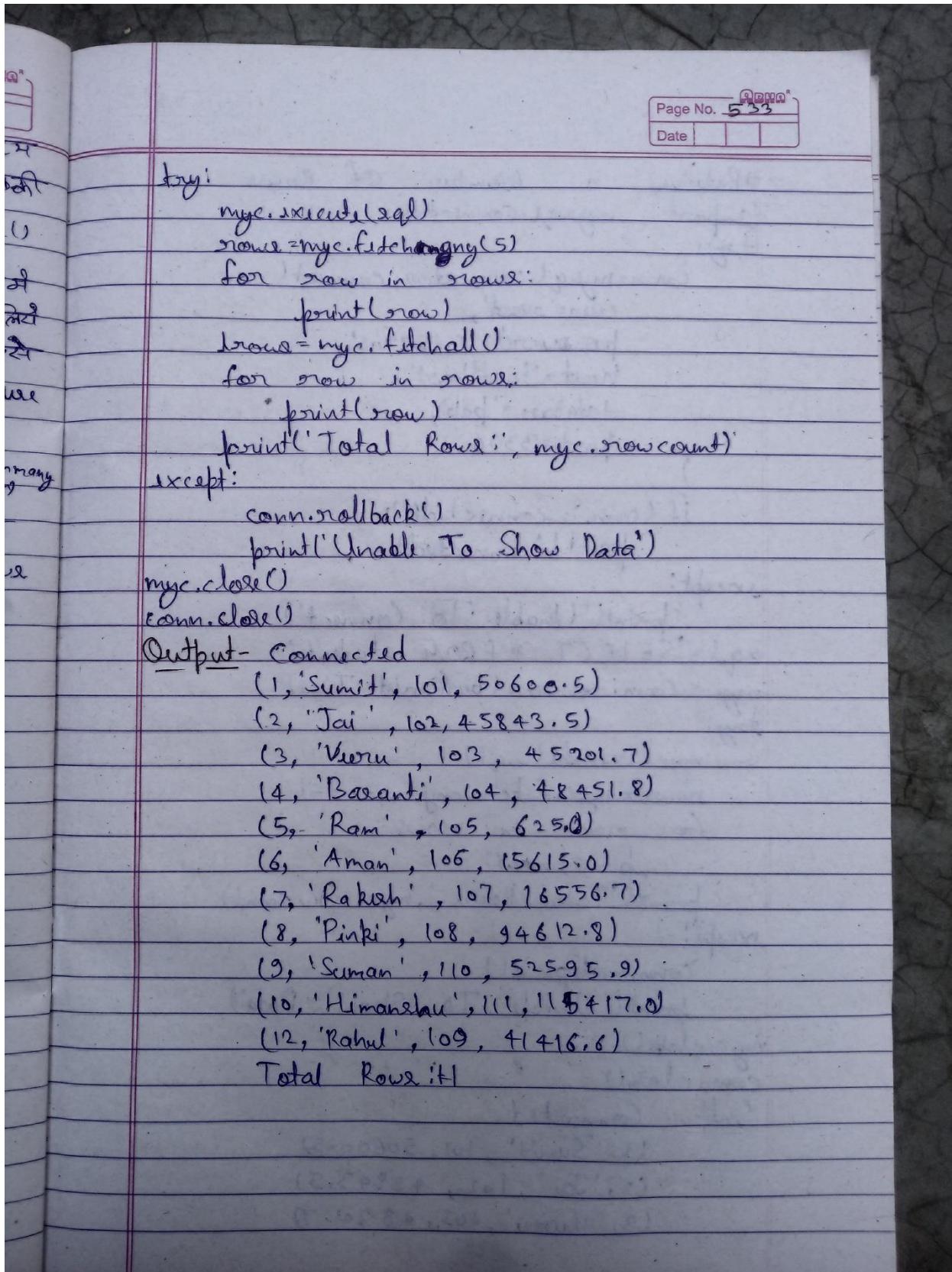
Page No. 532

Date

जबकि इसे problem का उल्लंघन करने की सहायता करती है। cursor को close करने से पहले तभी वह अपने show की fetchall() या fetchone() की वजह से read करके तब cursor की जांच close के बजाए पर इसमें मापदण्डित रूप से read करना यह होता है। इसलिए यहाँ cursor की जांच करना यह होता है। इसलिए यहाँ cursor (buffered = True) का use करने हैं अपना cursor बनाने के लिये आवश्यक cursor ने show read कर लेगा। परं फिरना show read करने के लिये फिर भी नहीं होगा। उतना ही read करने देगा। और इस situation में rowscount() all number of rows ही ~~not~~ return करता है।

→ # Retrieve n Number Of Rows

```
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
sql='SELECT * FROM student'
myc=conn.cursor()
```



Page No. 534 ADUNA  
Date \_\_\_\_\_

→ # Retrieve n Number of Rows

```

import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
    sql='SELECT * FROM student'
    myc=conn.cursor(buffered=True)
    try:
        myc.execute(sql)
        rows=myc.fetchmany(size=5)
        for row in rows:
            print(row)
        print('Total Rows:',myc.rowcount)
    except:
        conn.rollback()
        print('Unable To Show Data')
    myc.close()
    conn.close()

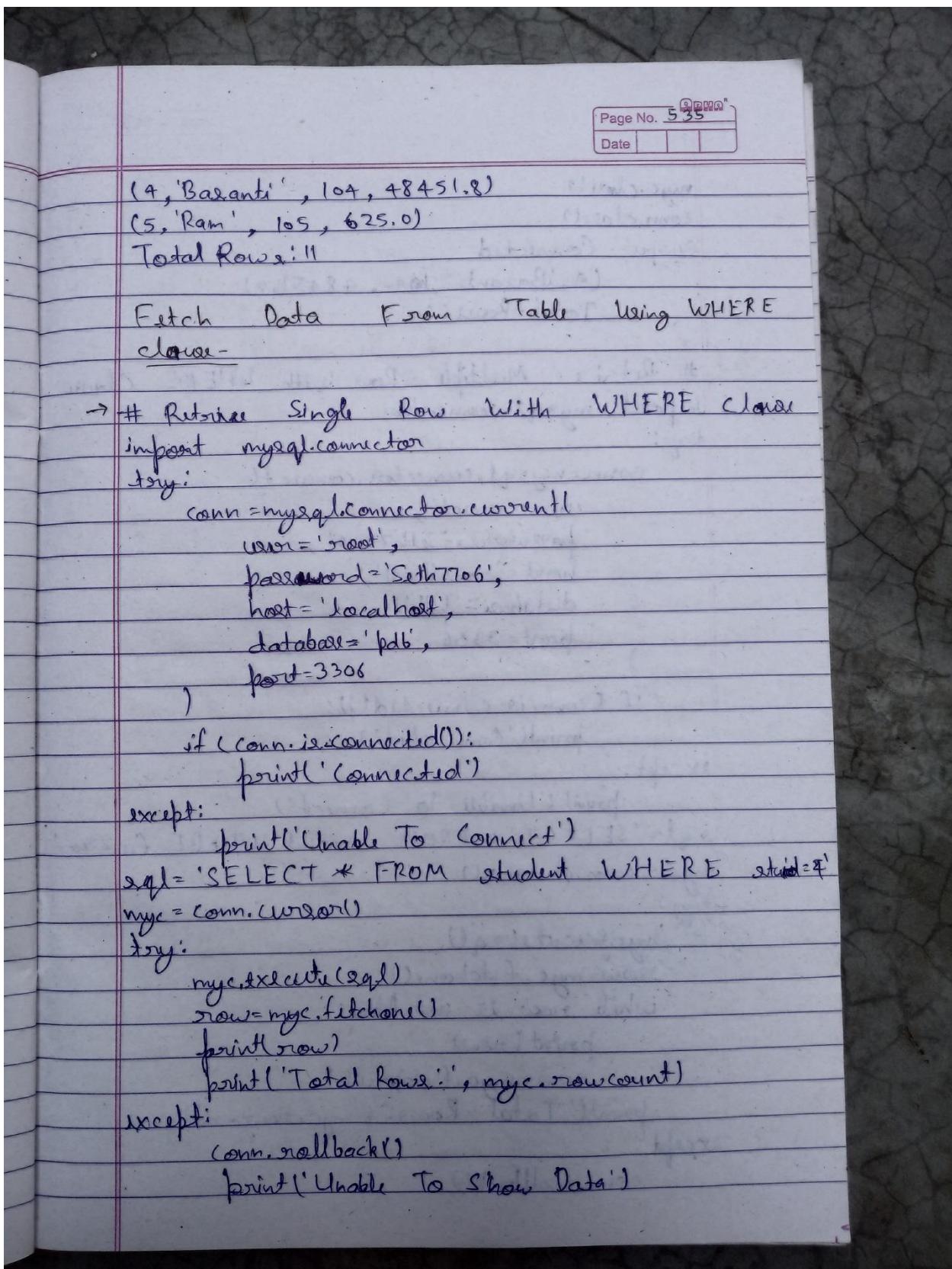
```

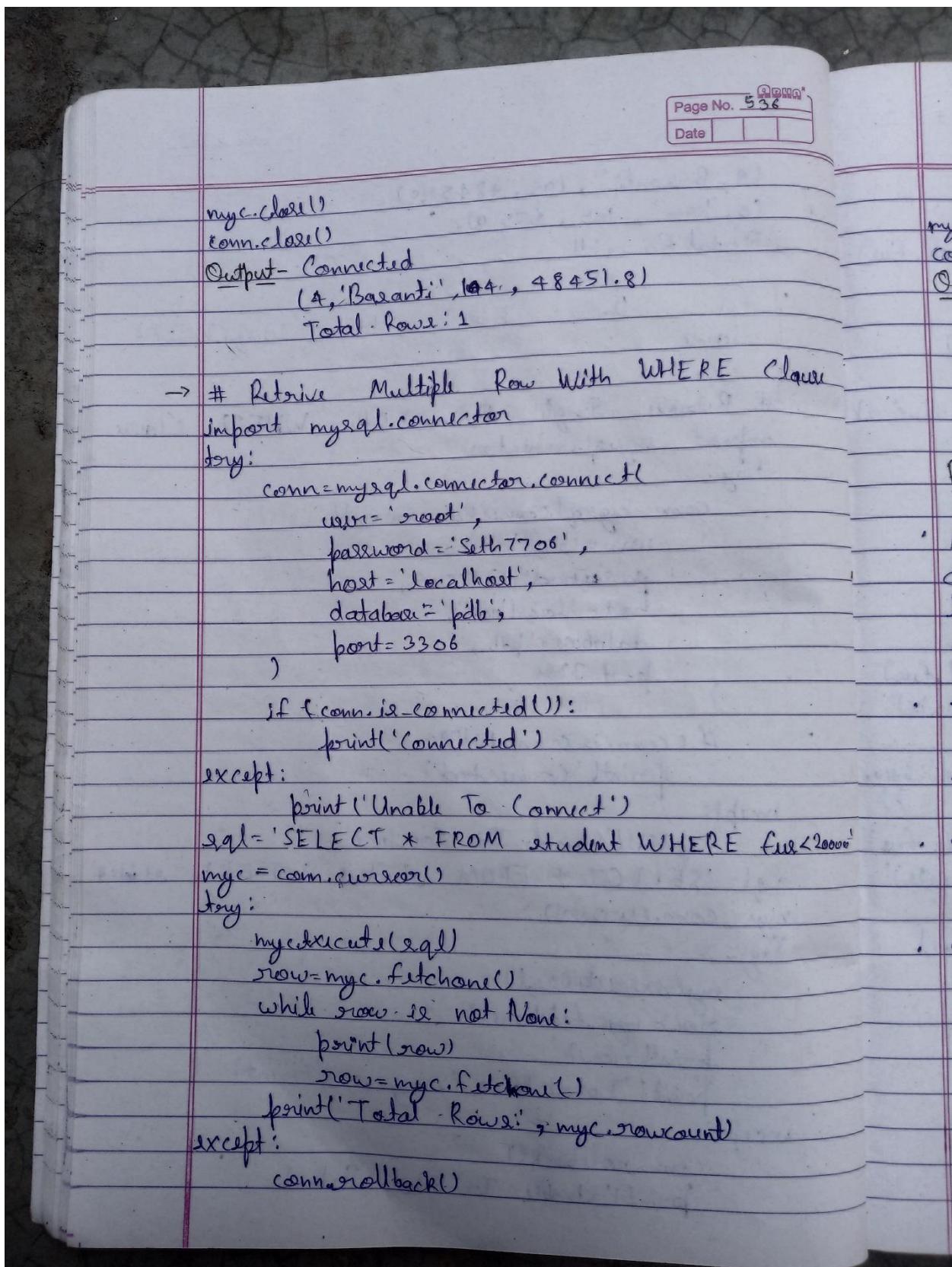
Output- Connected

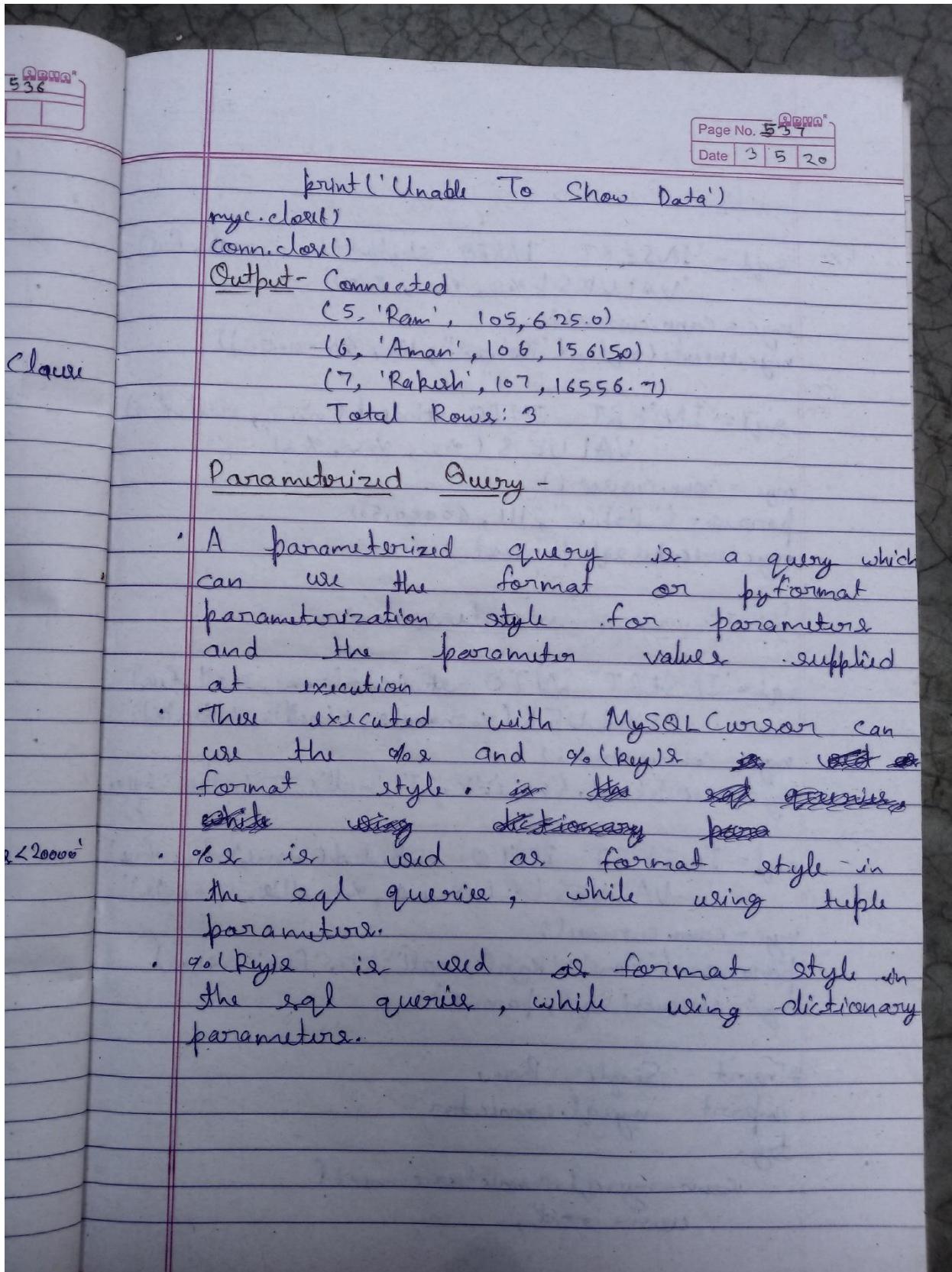
```

(1, 'Sumit', 101, 50600.5)
(2, 'Jai', 102, 45843.5)
(3, 'Kruni', 103, 45201.7)

```







Page No. 538  
Date

### Tuple Parameters -

Ex -  
`sql = 'INSERT INTO student (name, roll, fee)  
VALUES (%s, %s, %s)'  
myc = conn.cursor()  
myc.execute(sql, ("Rohan", 111, 60000.50))`

Or  
`sql = 'INSERT INTO student (name, roll, fee)  
VALUES (%s, %s, %s)'  
myc = conn.cursor()  
params = ("Rohan", 111, 60000.50)  
myc.execute(sql, params)`

### Dictionary Parameters -

`sql = 'INSERT INTO student (name, roll, fee)  
VALUES (%(name)s, %(roll)s, %(fee)s)'  
myc = conn.cursor()  
myc.execute(sql, {'name': 'Kajal', 'roll': 777, 'fee': 5400})`

Or  
`sql = 'INSERT INTO student (name, roll, fee)  
VALUES (%(name)s, %(roll)s, %(fee)s)'  
myc = conn.cursor()  
params = {'name': 'Kajal', 'roll': 777, 'fee': 5400}  
myc.execute(sql, params)`

→ # Input Single Row  
`import mysql.connector  
try:`

`conn = mysql.connector.connect(  
user = 'root',`

Page No. 539 Date

```

password='Seth7706',
host='localhost',
database='pdb',
port=3306
)
if (conn.is_connected()):
    print('Connected')
else:
    print('Unable To Connect')
sql='INSERT INTO student(name,roll,fee)
VALUES (%s,%s,%s)'
myc=conn.cursor()
try:
    myc.execute(sql,('Sumit',101,30000.50))
    conn.commit() # Committing The Change
    print('Total Rows:',myc.rowcount) # Number
    of Row Inserted
    print(f'Stu ID: {myc.lastrowid} Inserted')
    # Last Inserted ID
except:
    conn.rollback() # Rollback The Change
    print('Unable To Insert Data')
    myc.close() # Close Cursor
    conn.close() # Close Connection.
Output- Connected
Total Rows: 1
Stu ID: 1 Inserted

```

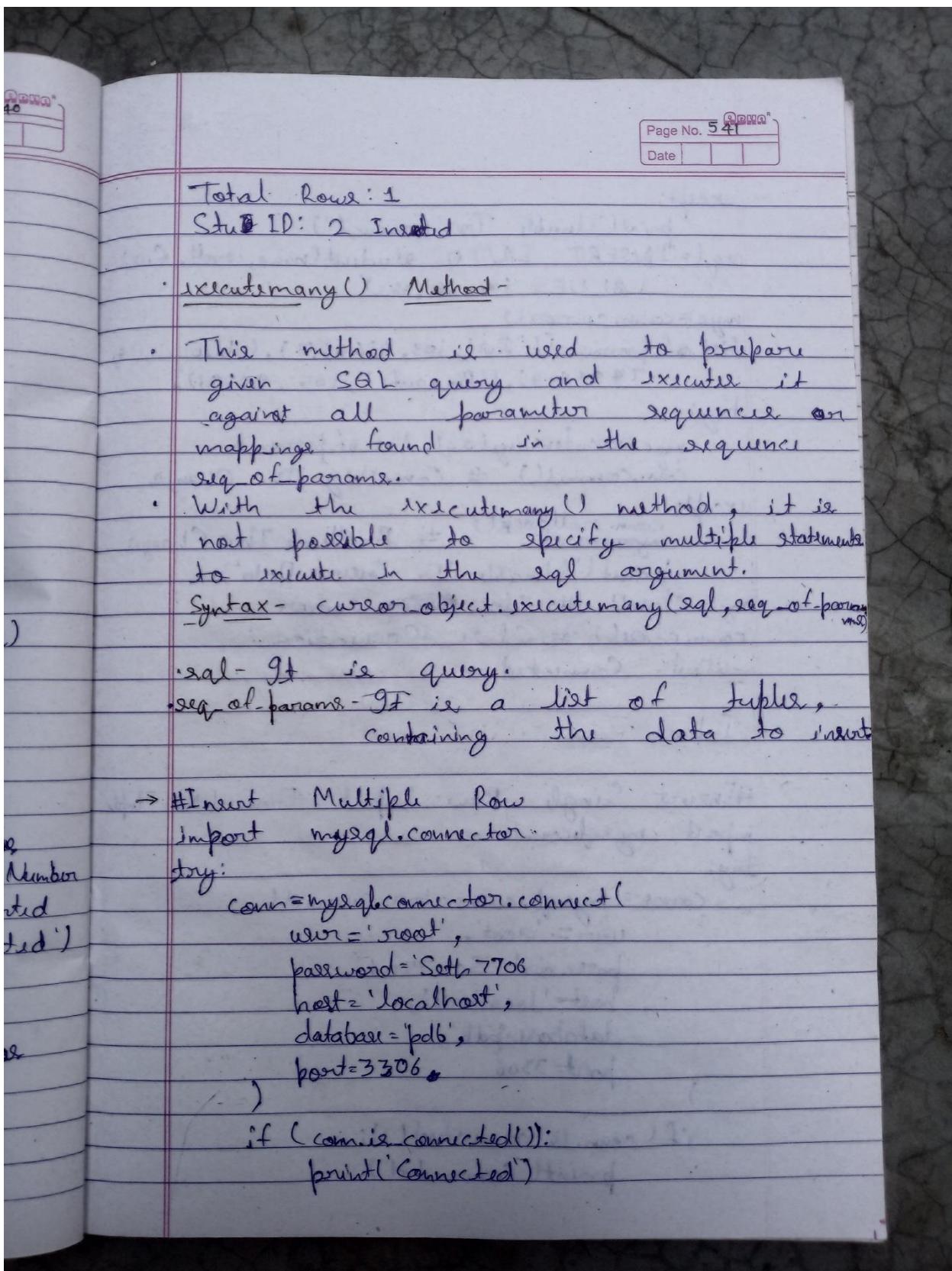
Page No. 540

Date

```

→ # Insert Single Row
import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if conn.is_connected():
        print('Connected')
    except:
        print('Unable To Connect')
    sql = 'INSERT INTO student(name, roll, fee) VALUES (%s, %s, %s)'
    myc = conn.cursor()
    params = ('Rani', 102, 35264.50)
    try:
        myc.execute(sql, params)
        conn.commit() # Committing The Changes
        print(f'Total Rows: {myc.rowcount} # Number of Row Inserted')
        print(f'Stu ID: {myc.lastrowid} Inserted') # Last Inserted ID
    except:
        conn.rollback() # Rollback The Changes
        print('Unable To Insert Data')
    myc.close() # Close Cursor
    conn.close() # Close Connection.
    Output - Connected

```



Page No. 542  
Date \_\_\_\_\_

```

except:
    print('Unable To Connect')
sql = "INSERT INTO student(name, roll, fee)
      VALUES (%s, %s, %s)"
myc = conn.cursor()
list_of_params = [('Jai', 103, 48646.7), ('Kruni', 104,
74844.4), ('Bazanti', 105, 98521)]
try:
    myc.executemany(sql, list_of_params)
    conn.commit() # Committing The Change
except:
    conn.rollback() # Rollback The Change
    print('Unable To Insert Data')
myc.close() # Close cursor
conn.close() # Close Connection
Output - Connected

```

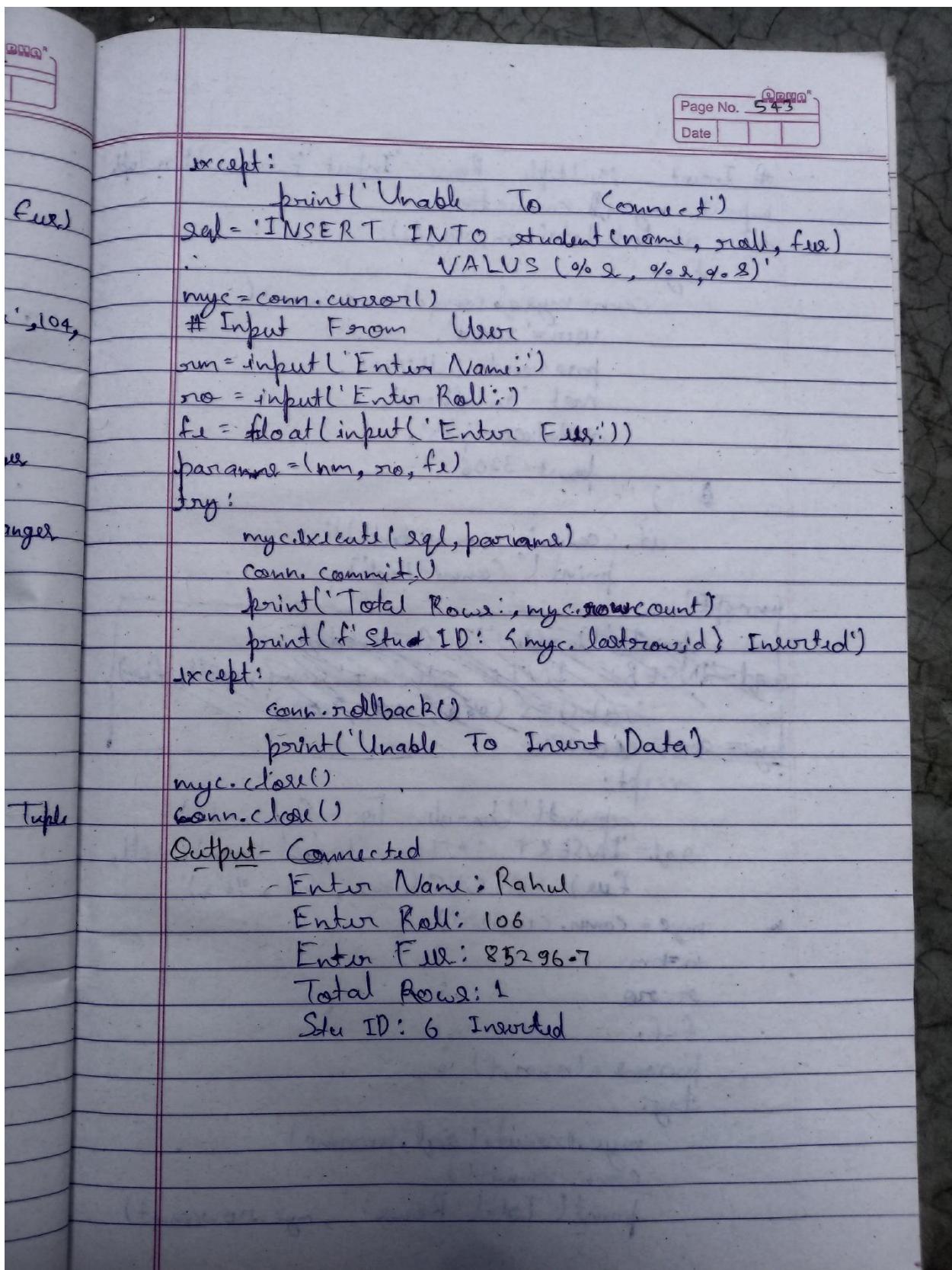
Input From User-

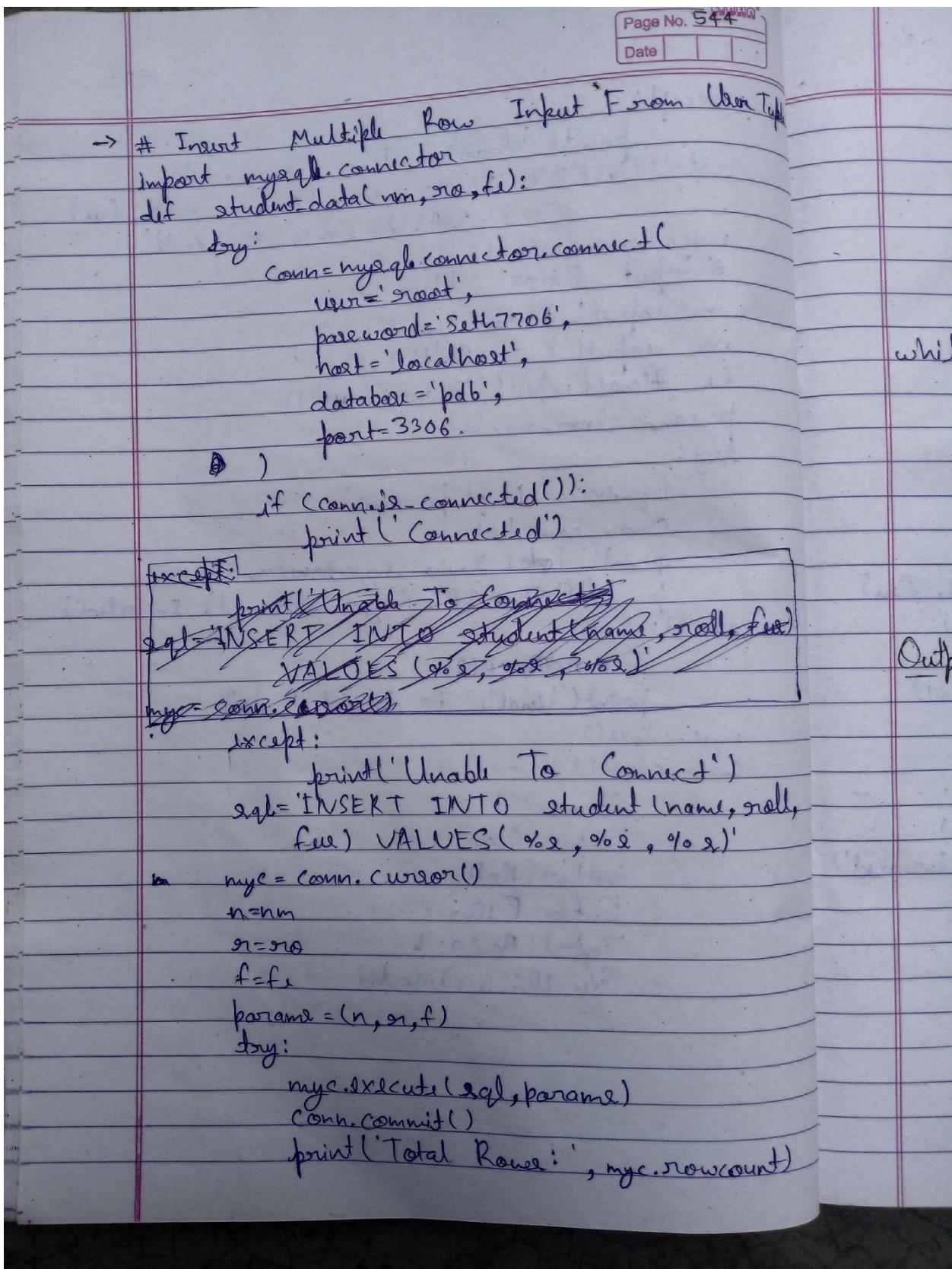
→ #Insert Single Row Input From User Tuple

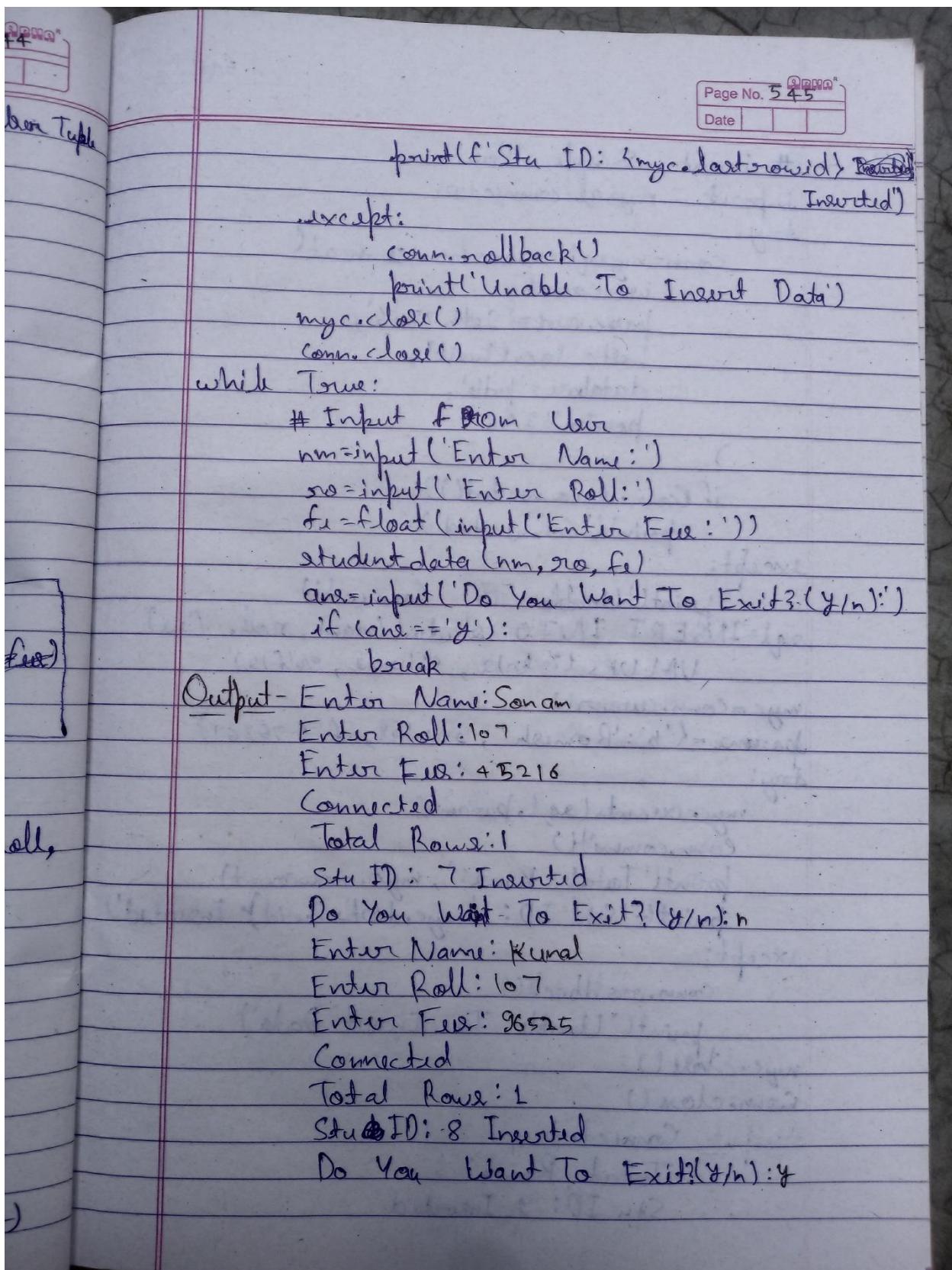
```

import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')

```





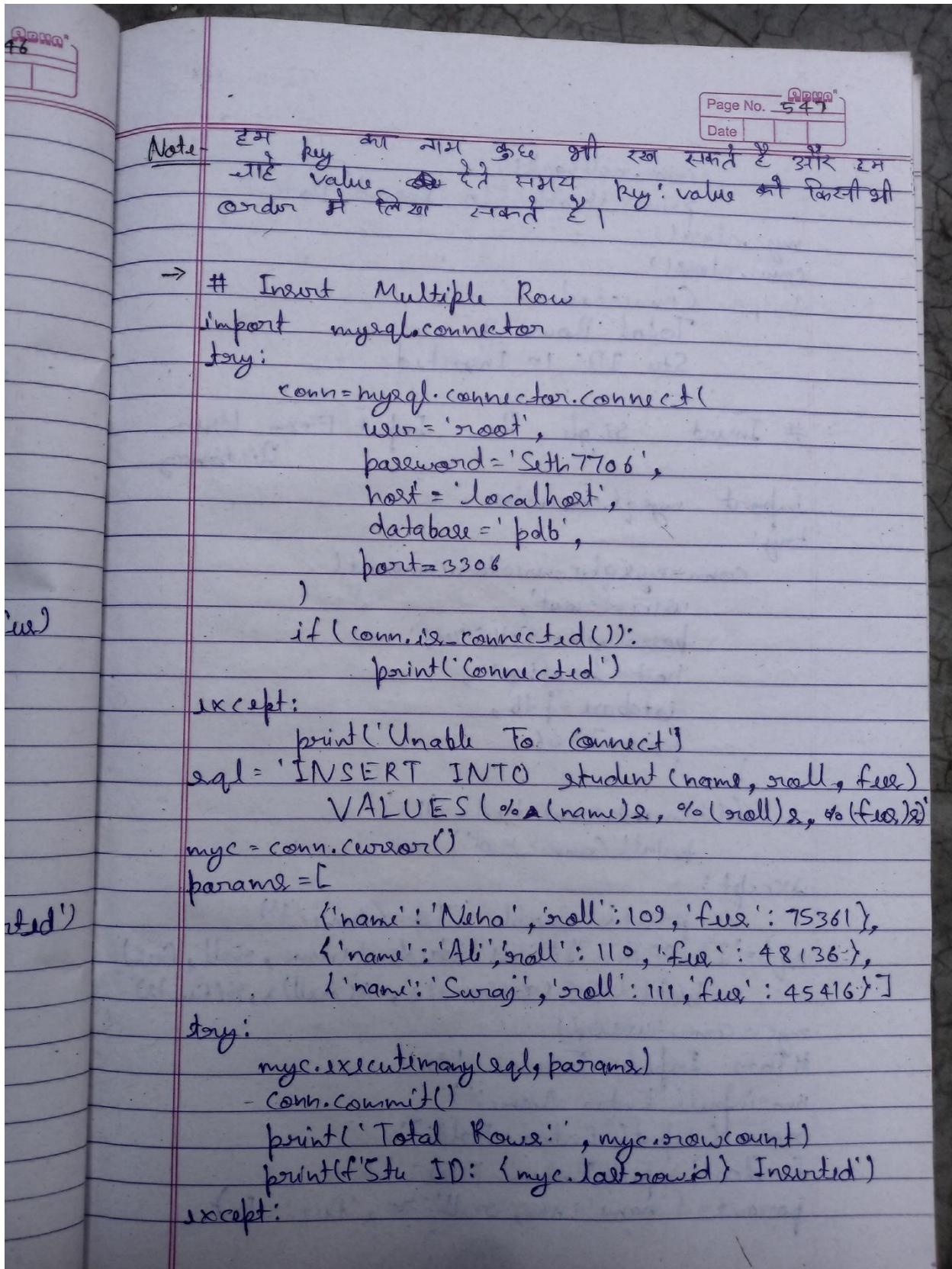


Page No. 546

```

→ # Insert Single Row
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
sql='INSERT INTO student(name, roll, fee)
VALUES (%(n)s, %(r)s, %(f)s)'
myc=conn.cursor()
params={'n':'Ramesh', 'r':108, 'f':7536}
try:
    myc.execute(sql,params)
    conn.commit()
    print('Total Rows:', myc.rowcount)
    print(f'Stu ID: {myc.lastrowid} Inserted')
except:
    conn.rollback()
    print('Unable To Insert Data')
myc.close()
conn.close()
Output- Connected
Total Rows: 1
Stu ID: 9 Inserted

```



Page No. 548

Date

```

conn.rollback()
print('Unable To Insert Data')
myc.close()
conn.close()
Output - Connected
Total Row: 3
Stu ID: 10 Inserted

```

→ # Insert Single Row Input From User Dictionary

```

import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if(conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
sql='INSERT INTO student(name,roll,fee)
VALUES (%(name)s,%(roll)s,%(fee)s)'
myc=conn.cursor()
# Data Input From User
nm=input('Enter Name:')
ro=int(input('Enter Roll:'))
fe=float(input('Enter Fee:'))
params={'name':nm,'roll':ro,'fee':fe}

```

Page No. 549  
Date

```

try:
    myc.execute(sql, param)
    conn.commit()
    print('Total Rows:', myc.rowcount)
    print(f'Stu ID: {myc.lastrowid} Inserted')
except:
    conn.rollback()
    print('Unable To Insert Data')
    myc.close()
    conn.close()

Output - Connected
Enter Name:
Enter Roll:
Enter Fee:
Total Rows: 1
Stu ID: 13 Inserted

```

→ # Insert Multiple Row Input From User Dictionary

```

import mysql.connector
def student_data(nm, ro, fe):
    try:
        conn = mysql.connector.connect(
            user='root',
            password='Seth7706',
            host='localhost',
            database='pdb',
            port=3306
        )
        if (conn.is_connected()):
            print('Connected')
    
```

Page No. 550  
Date: \_\_\_\_\_

```

except:
    print('Unable To Connect')
    exit()

INSERT INTO student(name, roll, fee)
VALUES (%(name)s, %(roll)s, %(fee)s)

myc = conn.cursor()
nm = nm
nr = nr
f = fe
params = {'name': nm, 'roll': nr, 'fee': f}

try:
    myc.execute(sql, params)
    conn.commit()
    print('Total Rows:', myc.rowcount)
    print(f'Student ID: {myc.lastrowid} Inserted')

except:
    conn.rollback()
    print('Unable To Insert Data')
    myc.close()
    conn.close()

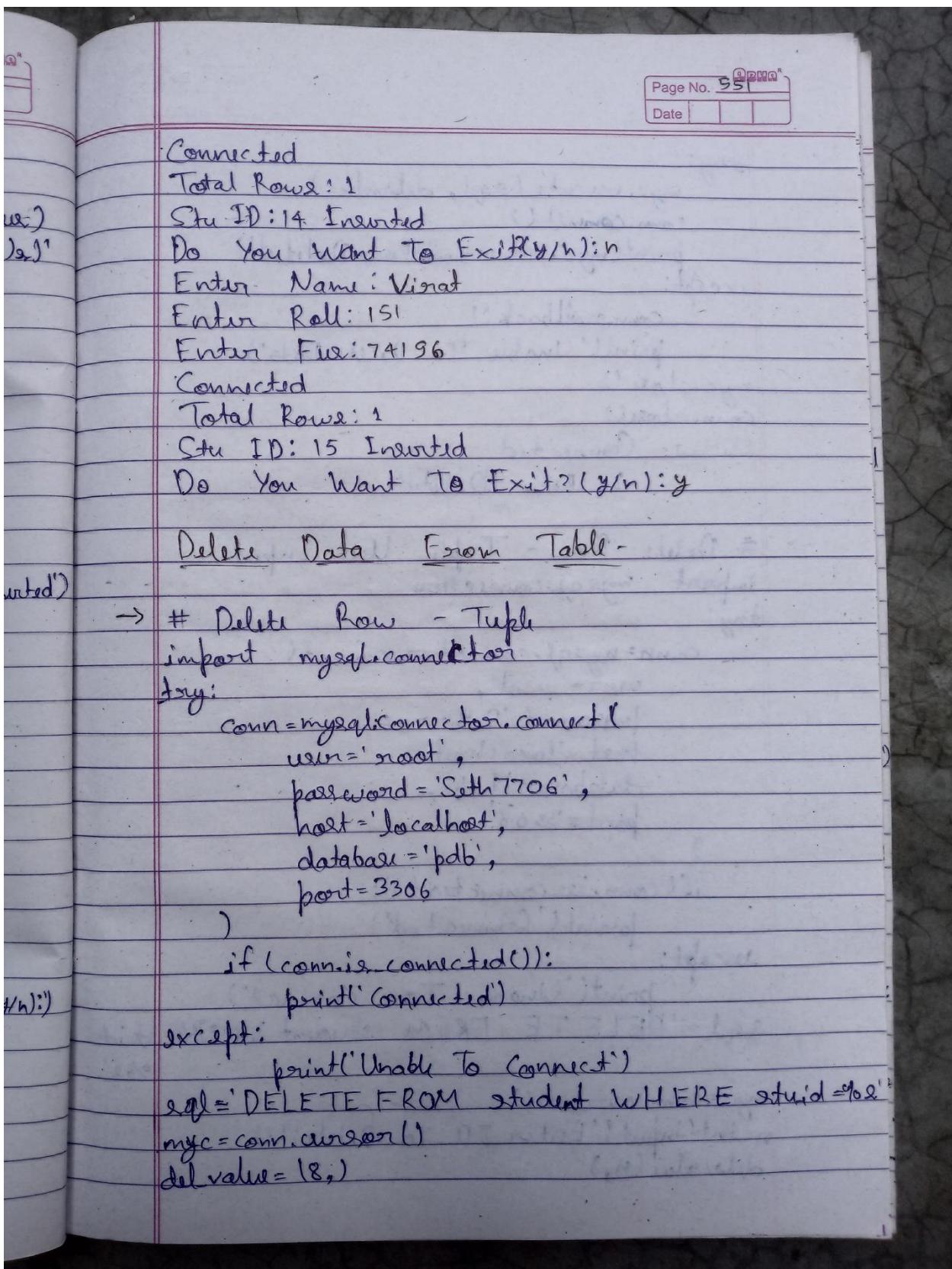
while True:
    # Data Input From User
    nm = input('Enter Name: ')
    no = int(input('Enter Roll: '))
    fe = float(input('Enter Fee: '))

    student_data(nm, no, fe)

    ans = input('Do You Want To Exit?(y/n): ')
    if ans == 'y':
        break

Output- Enter Name: Sahil
        Enter Roll: 123
        Enter Fee: 85296

```



Page No. 552

try:

```
myc.execute(sql, delvalue)
conn.commit()
print(myc.rowcount, "Row Deleted")
```

except:

```
conn.rollback()
print('Unable To Delete Data')
myc.close()
conn.close()
```

Output - Connected

1 Row Deleted

→ # Delete Row - Tuple User Input →

```
import mysql.connector
```

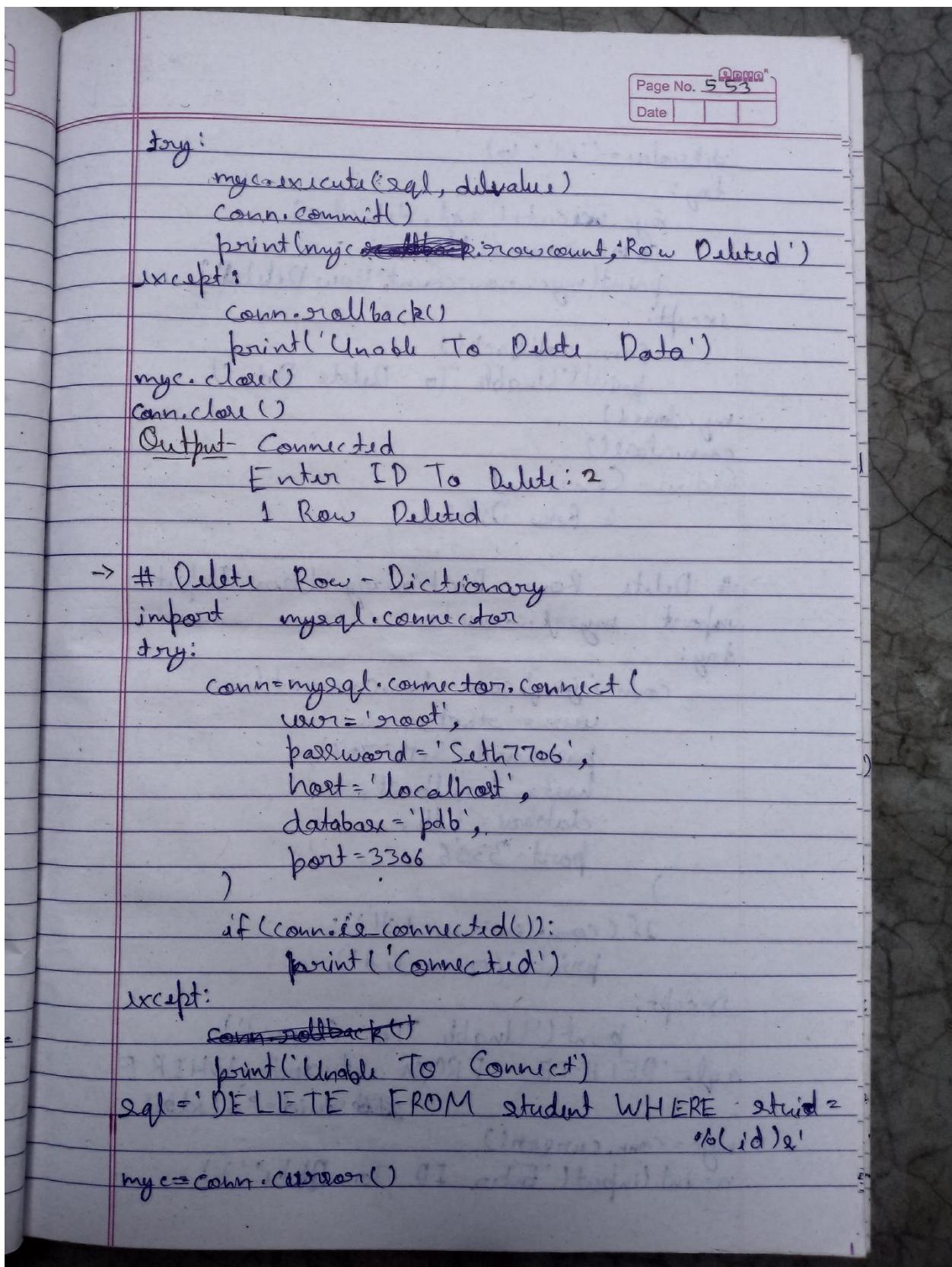
try:

```
conn = mysql.connector.connect(
    user='root',
    password='Seth7706',
    host='localhost',
    database='pdb',
    port=3306
)
if conn.is_connected():
    print('Connected')
```

except:

```
print('Unable To Connect')
sql = 'DELETE FROM student WHERE stdid=%d'
```

```
myc = conn.cursor()
n = int(input('Enter ID To Delete:'))
delvalue(n,)
```



Page No. 554  
Date

```

dictvalue = {'id': 10}
try:
    myc.execute(sql, dictvalue)
    conn.commit()
    print(myc.rowcount, "Row Deleted")
except:
    conn.rollback()
    print("Unable To Delete Data")
my.close()
conn.close()

Output - Connected
1 Row Deleted

```

→ # Delete Row - Dictionary User Input

```

import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
    sql='DELETE FROM student WHERE
        studid=%(id)s'
    myc=conn.cursor()
    n=int(input('Enter ID To Delete'))

```

Page No. 555  
Date

```

del_value = { 'id': n }
try:
    myc.execute(sql, del_value)
    conn.commit()
    print(myc.rowcount, "Row Deleted")
except:
    conn.rollback()
    print('Unable To Delete Data')
myc.close()
conn.close()

```

Output - Connected

```

Enter ID To Delete : 11
1 Row Deleted

```

Update Date In Table -

```

→ # Update Row - Tuple
import mysql.connector
try:
    conn = mysql.connector.connect(
        user = 'sset',
        password = 'SethT106',
        host = 'localhost',
        database = 'pdb',
        port = 3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
sql = Update UPDATE student SET fme=90 WHERE
        admid = 90

```

Page No. 556  
Date

```

myc = conn.cursor()
update_value = (5000, 4)
try:
    myc.execute(sql, update_value)
    conn.commit()
    print(myc.rowcount, 'Row Updated')
except:
    conn.rollback()
    print('Unable To Update Data')
myc.close()
conn.close()

Output - Connected
1 Row Updated

```

→ # Update Row - Tuple User Input

```

import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
sql = 'UPDATE student SET name=%s, roll=%s,'
      'fees=%s WHERE studid=%s'
myc = conn.cursor()

```

Page No. 551

```

id = int(input('Enter Student ID To Update:'))
nm = input('Enter Name:')
ra = int(input('Enter Roll:'))
fe = float(input('Enter Fee:'))
updatevalue = (nm, ra, fe, id)
try:
    myc.execute(sql, updatevalue)
    conn.commit()
    print(myc.rowcount, 'Row Updated')
except:
    conn.rollback()
    print('Unable To Update Data')
myc.close()
conn.close()

Output - Connected
Enter Student ID To Update: 9
Enter Name: Jack
Enter Roll: 123 125
Enter Fee: 983.25
1 Row Updated

```

→ # Update Row - Dictionary

```

import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )

```

Page No. 558  
Date

```

if (conn.is_connected()):
    print('Connected')
except:
    print('Unable To Connect')
sql='UPDATE student SET name=%(nm)s,
        fee=%(fe)s WHERE stuid=%(id)s'
myc=conn.cursor()
update_value={'nm': 'Iron Man', 'fe': 700, 'id': 12}
try:
    myc.execute(sql, update_value)
    conn.commit()
    print(myc.rowcount, 'Row Updated')
except:
    conn.rollback()
    print(myc.rowcount, 'Row Updated')
    print('Unable To Update Data')
myc.close()
conn.close()
Output - Connected
    1 Row Updated
→ # Update Row - Dictionary User Input
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )

```

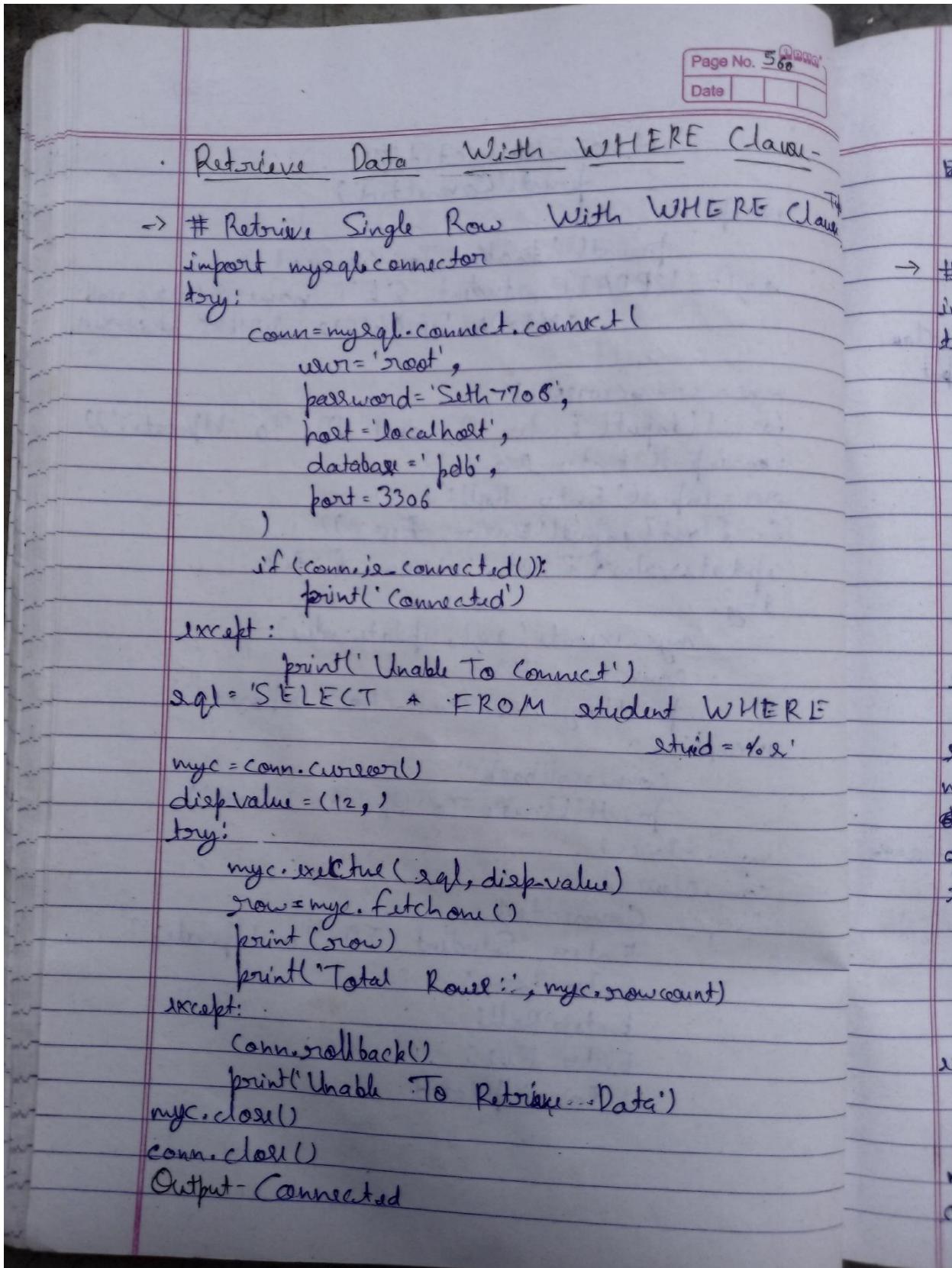
Page No. 559  
Date

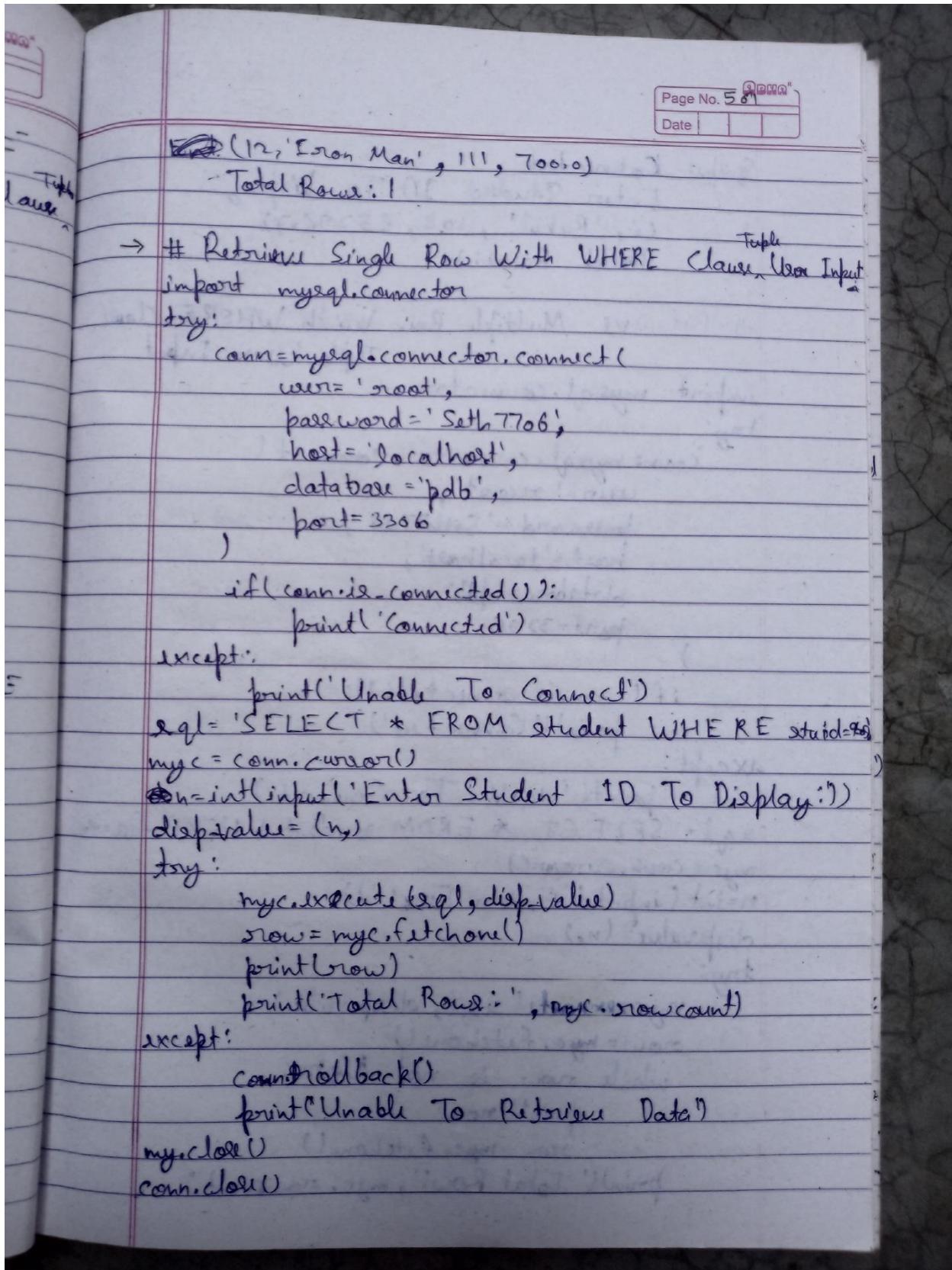
```

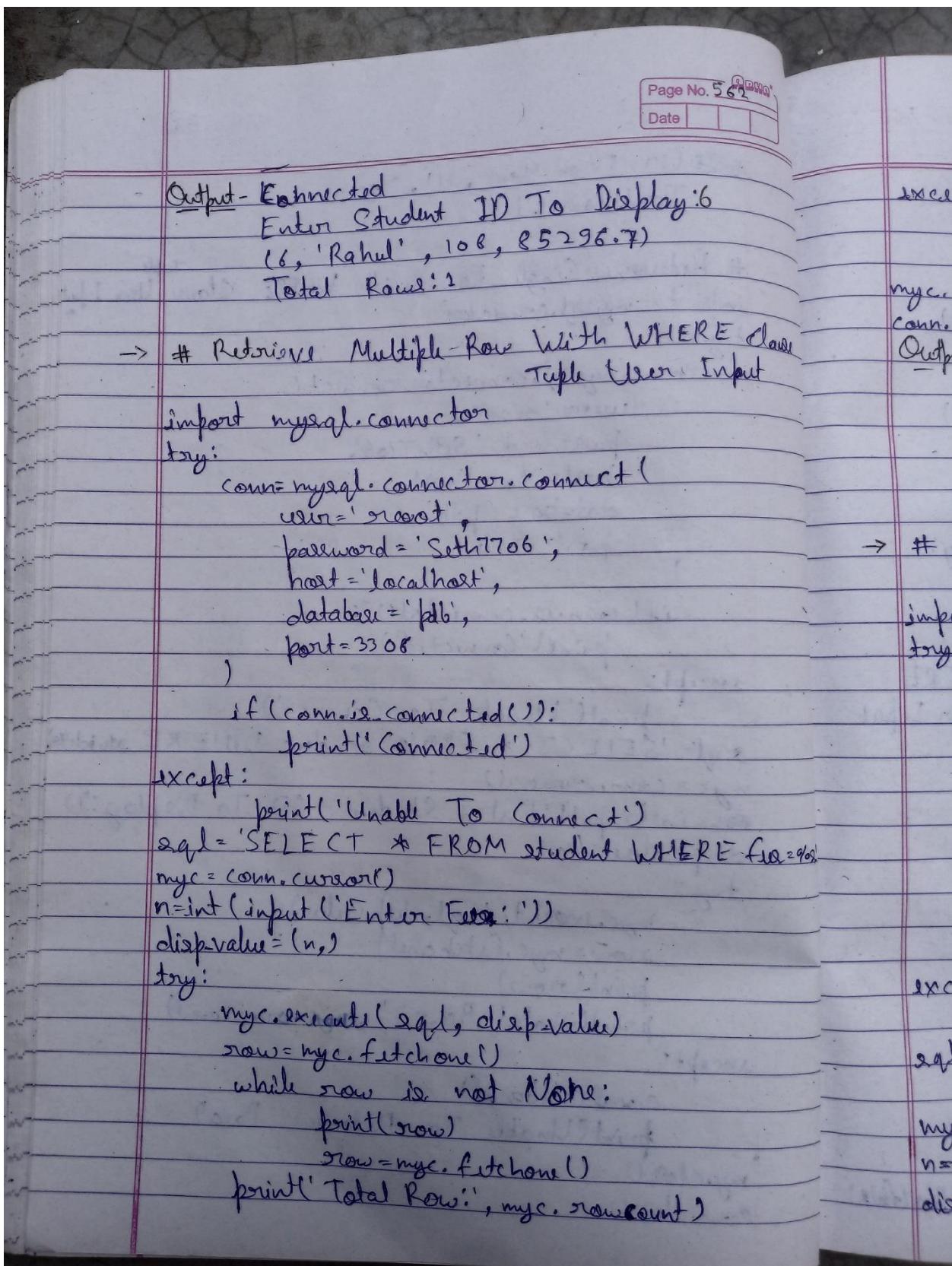
if (conn.isconnected()):
    print('Connected')
except:
    print('Unable To Connect')
sql = 'UPDATE student SET name=%(n)s, roll=%(r)s, fee=%(f)s WHERE studid=%(i)s'
myc = conn.cursor()
id = int(input('Enter Student ID To Update:'))
nm = input('Enter Name:')
rg = input('Enter Roll:')
fe = float(input('Enter Fee:'))
updatevalue = {'i': id, 'n': nm, 'f': fe}
try:
    myc.execute(sql, updatevalue)
    conn.commit()
    print(myc.rowcount, 'Row Updated')
except:
    conn.rollback()
    print('Unable To Update Data')
myc.close()
conn.close()

Output - Connected
Enter Student ID To Update:13
Enter Name:X Man
Enter Roll: 852
Enter Fee: 1002
1 Row Updated

```







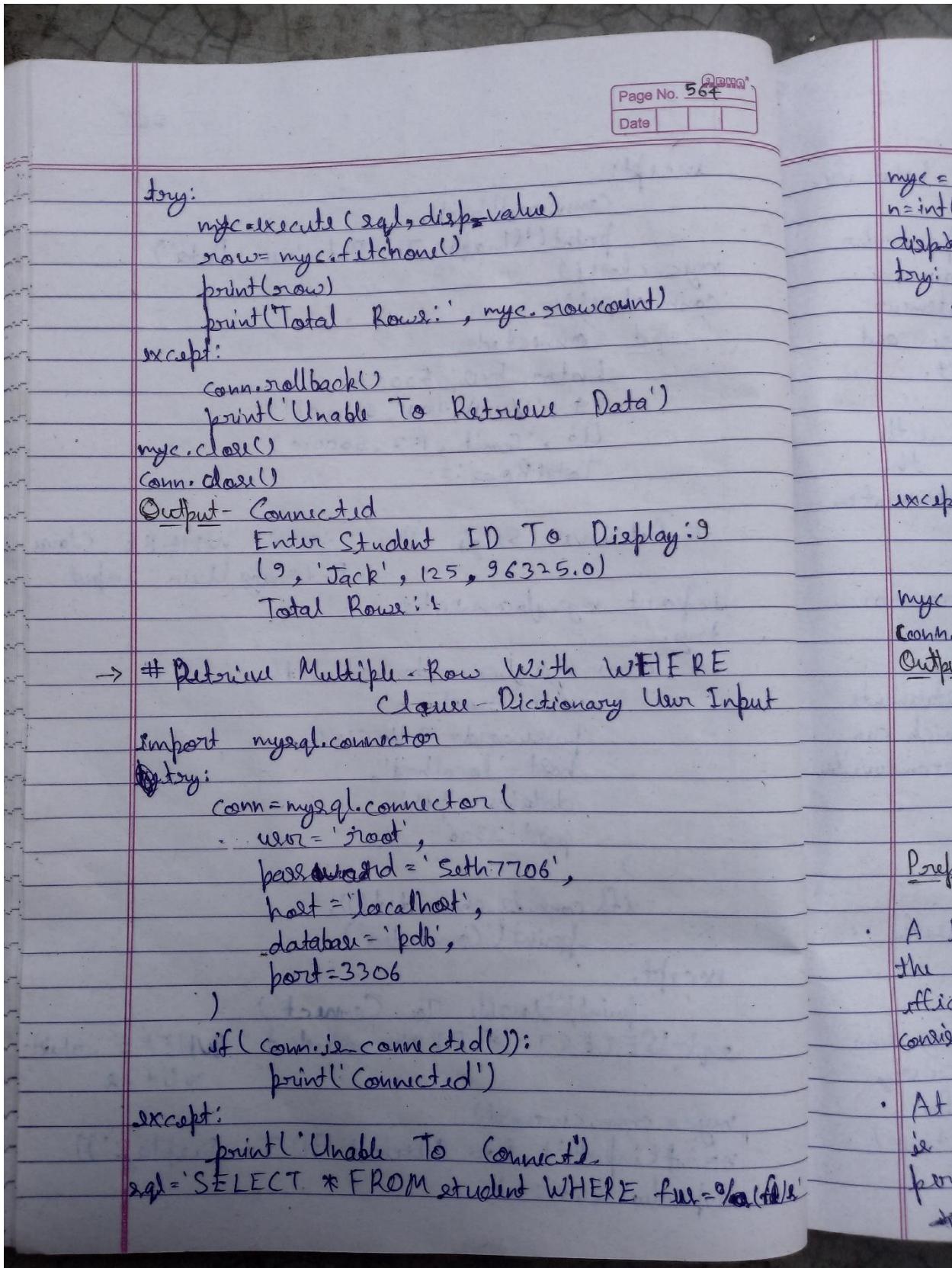
Page No. 563

except:  
 conn.rollback()  
 print('Unable To Retrieve Data')  
 myc.close()  
 conn.close()

lase  
Output - Connected  
 Enter Fee: 5000  
 (4, 'Kavita', 104, 5000.0)  
 (16, 'Sumit', 152, 5000.0)  
 Total Row: 2

→ # Retrieve Single Row With WHERE clause  
 Dictionary User Input

```
import mysql.connector
try:
    conn=mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if conn.is_connected():
        print('Connected')
except:
    print('Unable To Connect')
sql='SELECT * FROM student WHERE studid=%(id)s'
myc=conn.cursor()
n=int(input('Enter Student ID To Display:'))
displayvalue={'id':n}
```



Page No. 565 Date

```

myc = conn.cursor()
n = int(input('Enter File: '))
disp_value = {'fi': n}
try:
    myc.execute(sql, disp_value)
    row = myc.fetchone()
    while row is not None:
        print(row)
        row = myc.fetchone()
    print('Total Rows:', myc.rowcount)
except:
    conn.rollback()
    print('Unable To Retrieve Data')
    myc.close()
    conn.close()

```

Output - Connected

Enter File: 5000

```

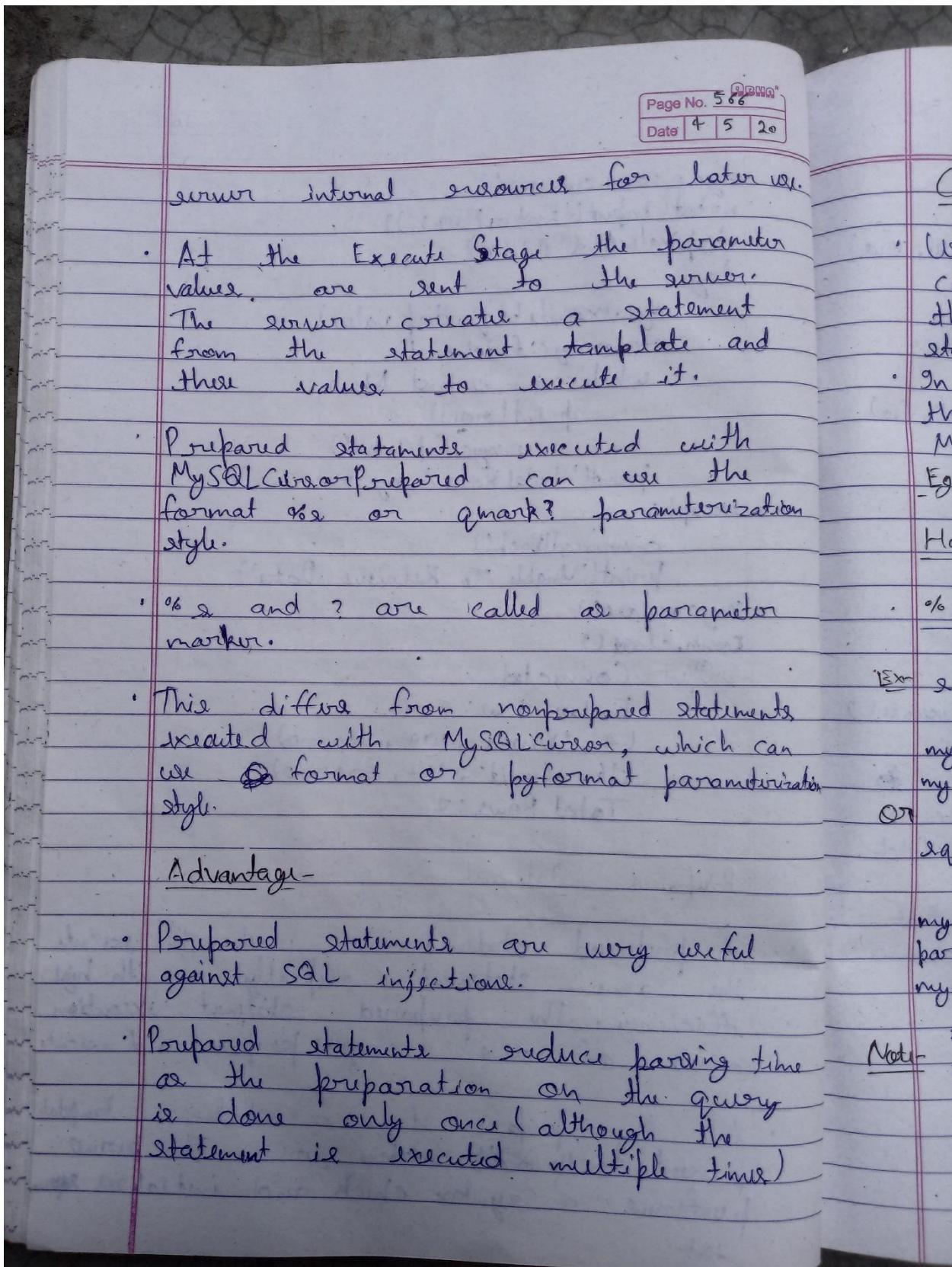
(4, 'Venu', 104, 50000.0)
(16, 'Sumit', 152, 50000.0)

```

Total Rows: 2

Prepared Statement-

- A prepared statement is used to execute the same statement repeatedly with high efficiency. The prepared statement execution consists of two stages: prepare and execute.
- At the prepare stage a statement template is sent to the ~~the~~ database server. The server performs a syntax check and initializes ~~the~~.



Page No. 561

Creating a Cursor -

- Using `prepared=True` argument to the `cursor()` method, creates a cursor that enables execution of prepared statements using the binary protocol.
- In this case, the `cursor()` method of the connection object returns a `MySQLCursorPrepared` object.

Eg:- `myc = conn.cursor(prepared=True)`

How to use -

%s parameter marker -

Ex:- `eql = 'INSERT INTO student (name, roll, fee)  
VALUES (%s, %s, %s)'`  
`myc = conn.cursor(prepared=True)`  
`myc.execute(eql, ('Rohan', 111, 60000.50))`

OR

`eql = 'INSERT INTO student (name, roll, fee)  
VALUES (%s, %s, %s)'`  
`myc = conn.cursor(prepared=True)`  
`params = ('Rohan', 111, 60000.50)`  
`myc.execute(eql, params)`

Note - ~~It is a tuple to use list & Dictionary~~

Page No. 568  
Date

- ? parameter marker-

Eg- `sqli = 'INSERT INTO student (name, roll, fee)  
VALUES (?, ?, ?)'`  
`myc = conn.cursor(prepared=True)`  
`myc.execute(sqli, ('Rohan', 111, 60000.50))`

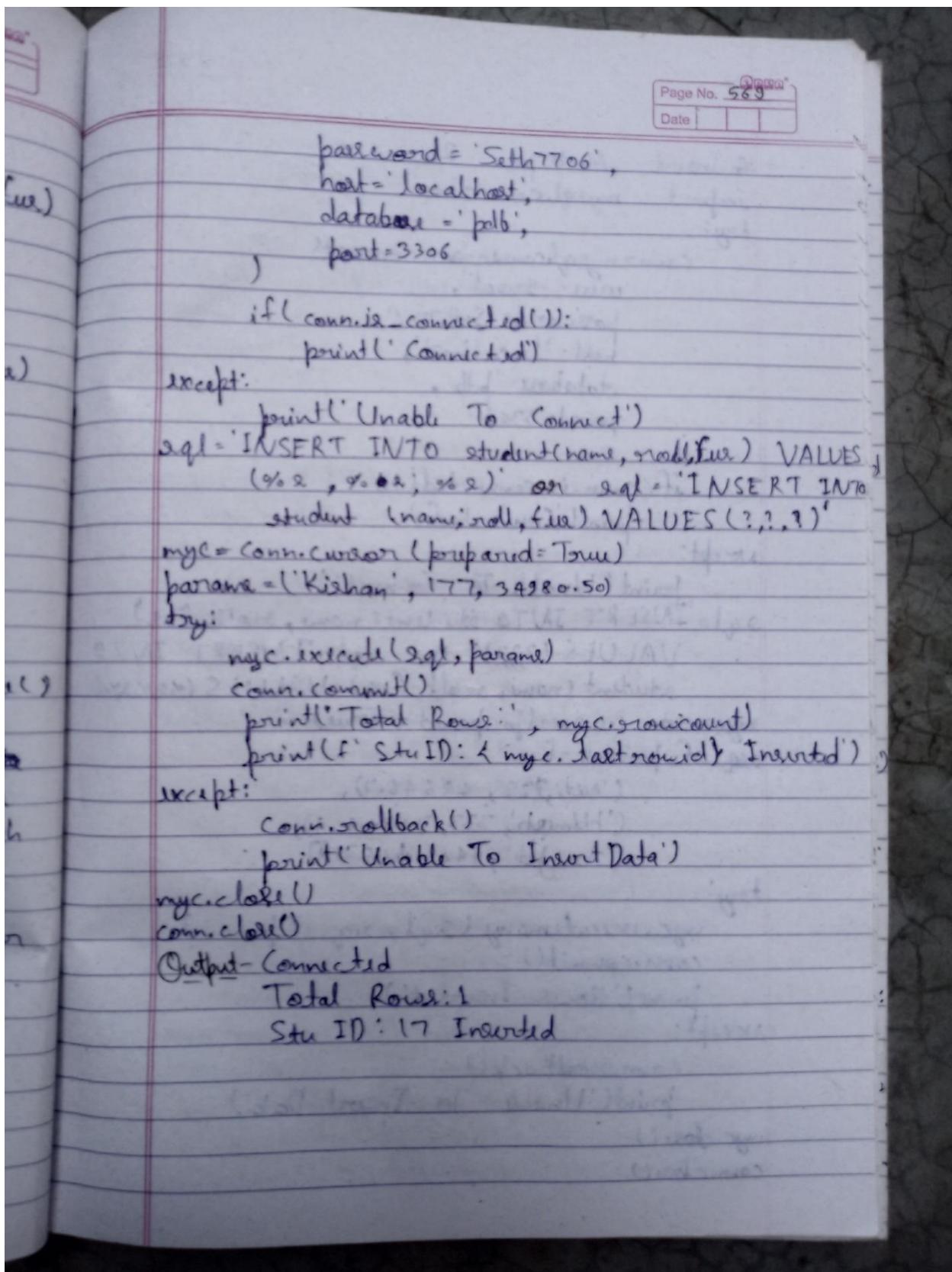
On `sqli = 'INSERT INTO student (name, roll, fee)  
VALUES (?, ?, ?)'`  
~~myc.execute~~  
`myc = conn.cursor(prepared=True)`  
`paname = ('Rohan', 111, 60000.50)`  
`myc.execute(sqli, paname)`

- How it Works-

- For the first call to the `execute()` method, the cursor prepares the statement. If data is given in the same call, it also executes the statement and you should fetch the data.
- For subsequent `execute()` calls that the same SQL statement, the cursor skips the preparation phase.

• Insert Row -

→ # Insert Single Row  
`import mysql.connector`  
`try:`  
`conn = mysql.connector.connect(  
 user='root',`

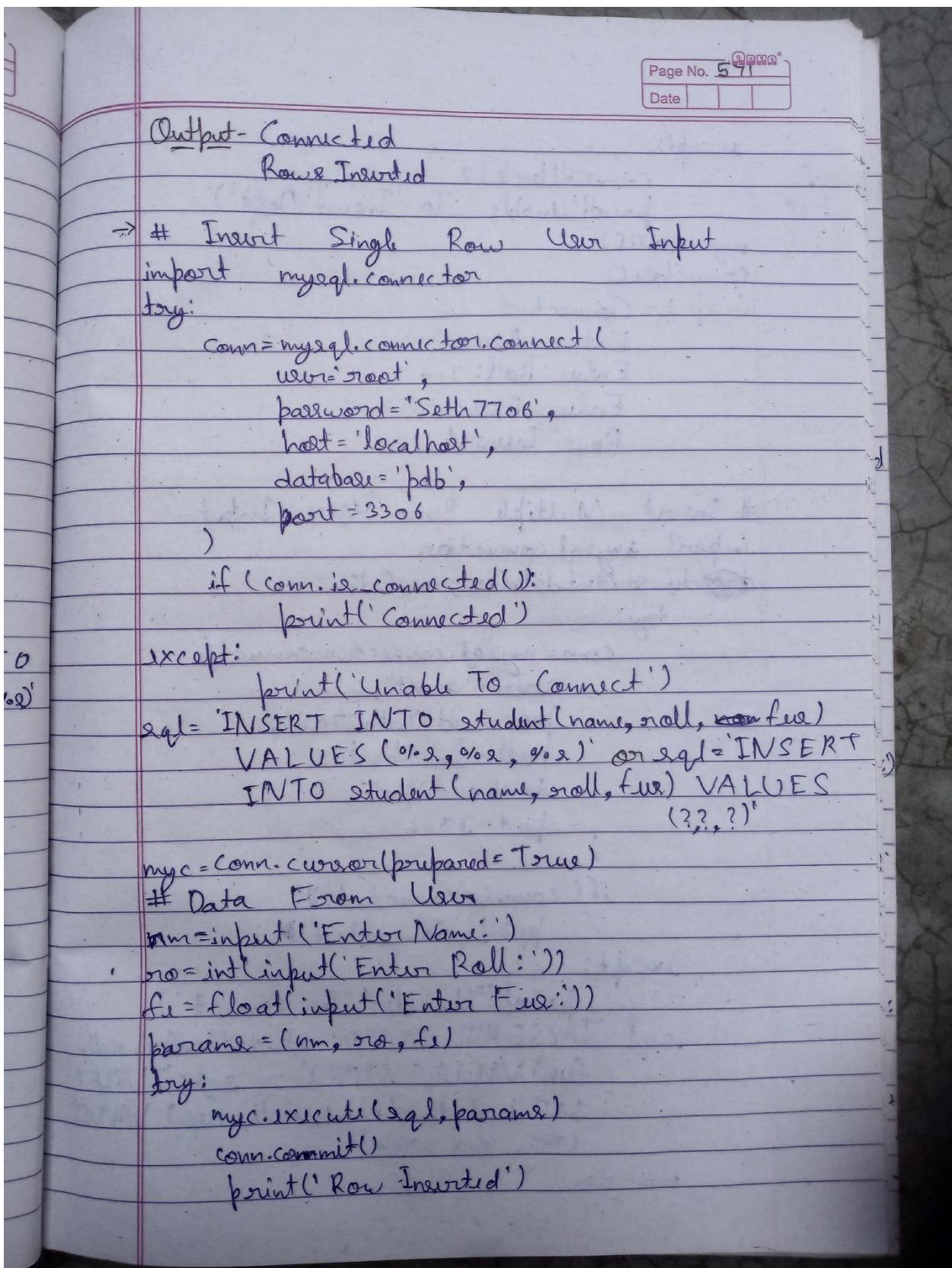


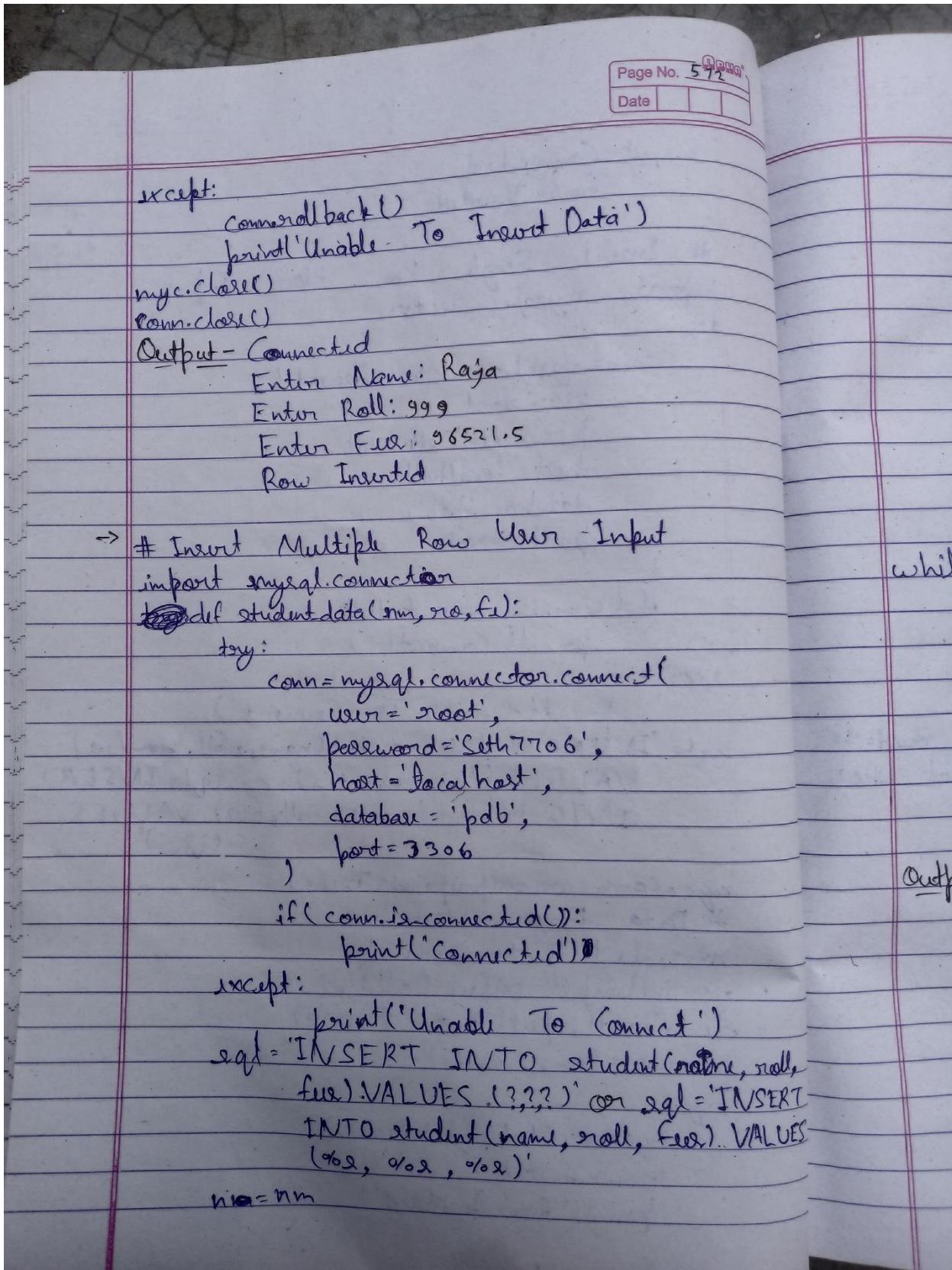
Page No. 570  
Date: \_\_\_\_\_

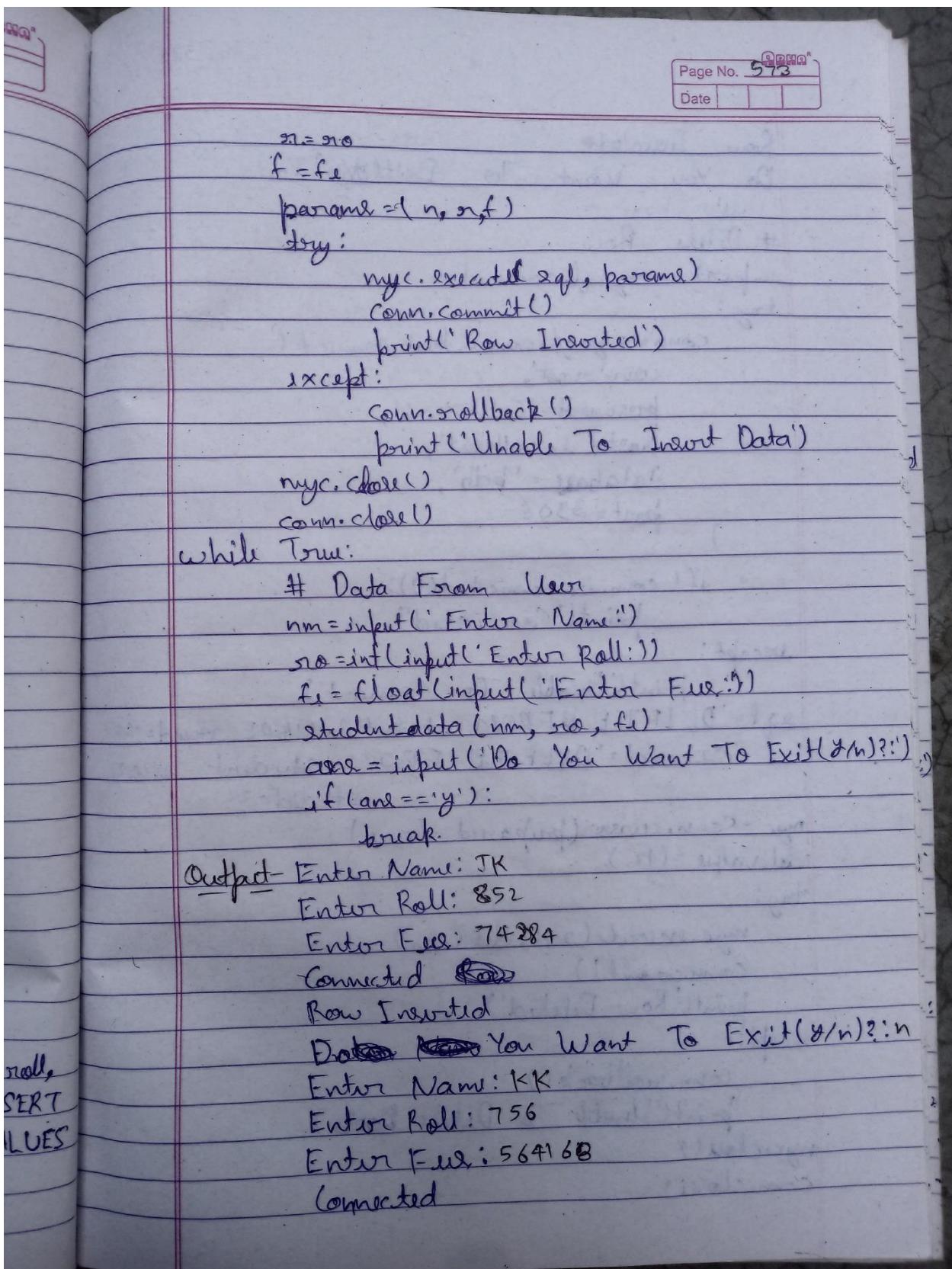
```

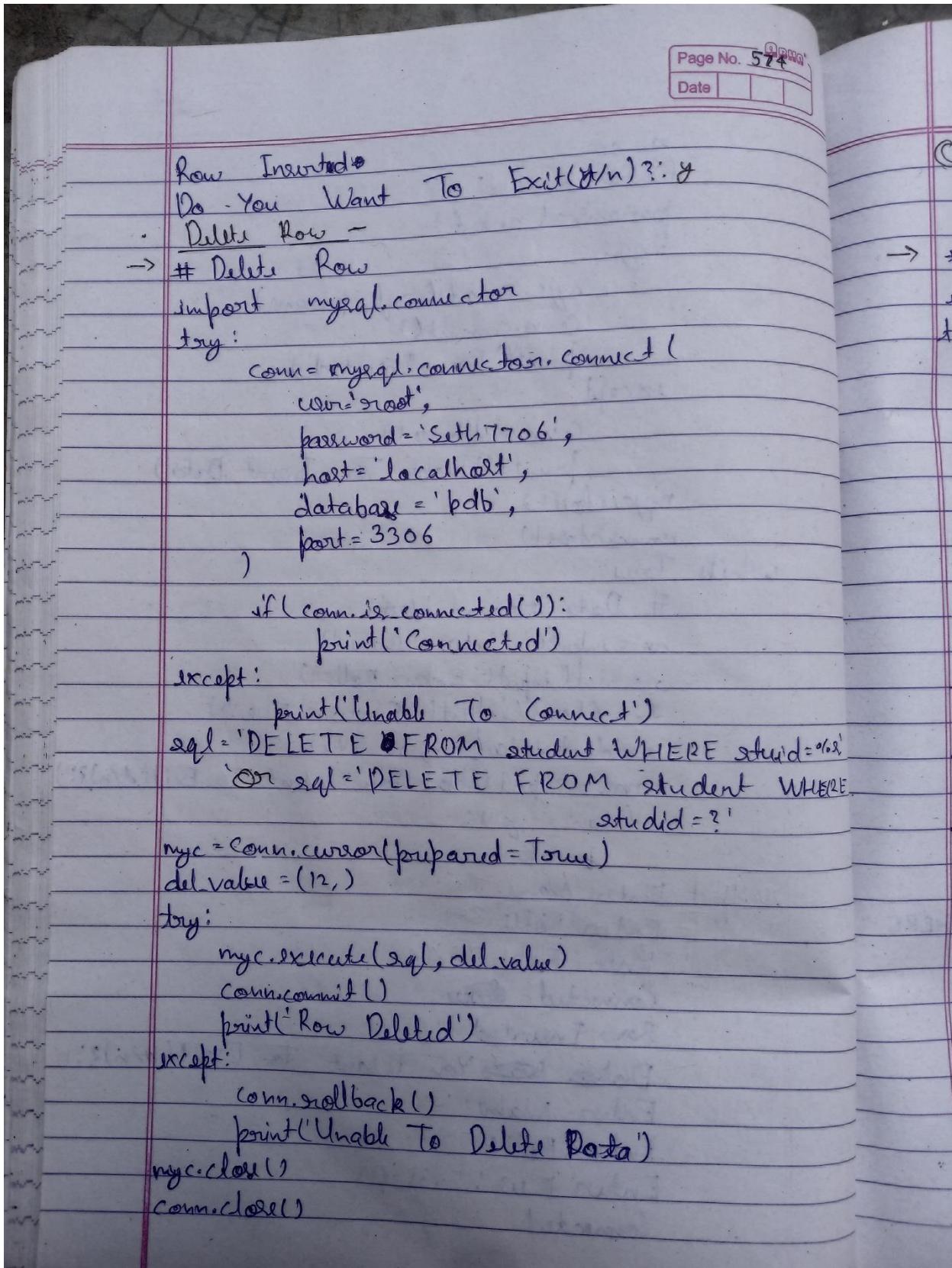
→ # Insert Multiple Row
import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='fpdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')
sql = 'INSERT INTO student(name, roll, fee)'
VALUES (?,?)' or sql = 'INSERT INTO
student(name, roll, fee) VALUES (%s,%s,%s)'
myc = conn.cursor(prepared=True)
seq_of_params = [
    ('Nuti', 222, 48646.7),
    ('Himeti', 333, 74844.4),
    ('Arijit', 444, 96521)]
try:
    myc.executemany(sql, seq_of_params)
    conn.commit()
    print('Rowe Inserted')
except:
    conn.rollback()
    print('Unable To Insert Data')
myc.close()
conn.close()

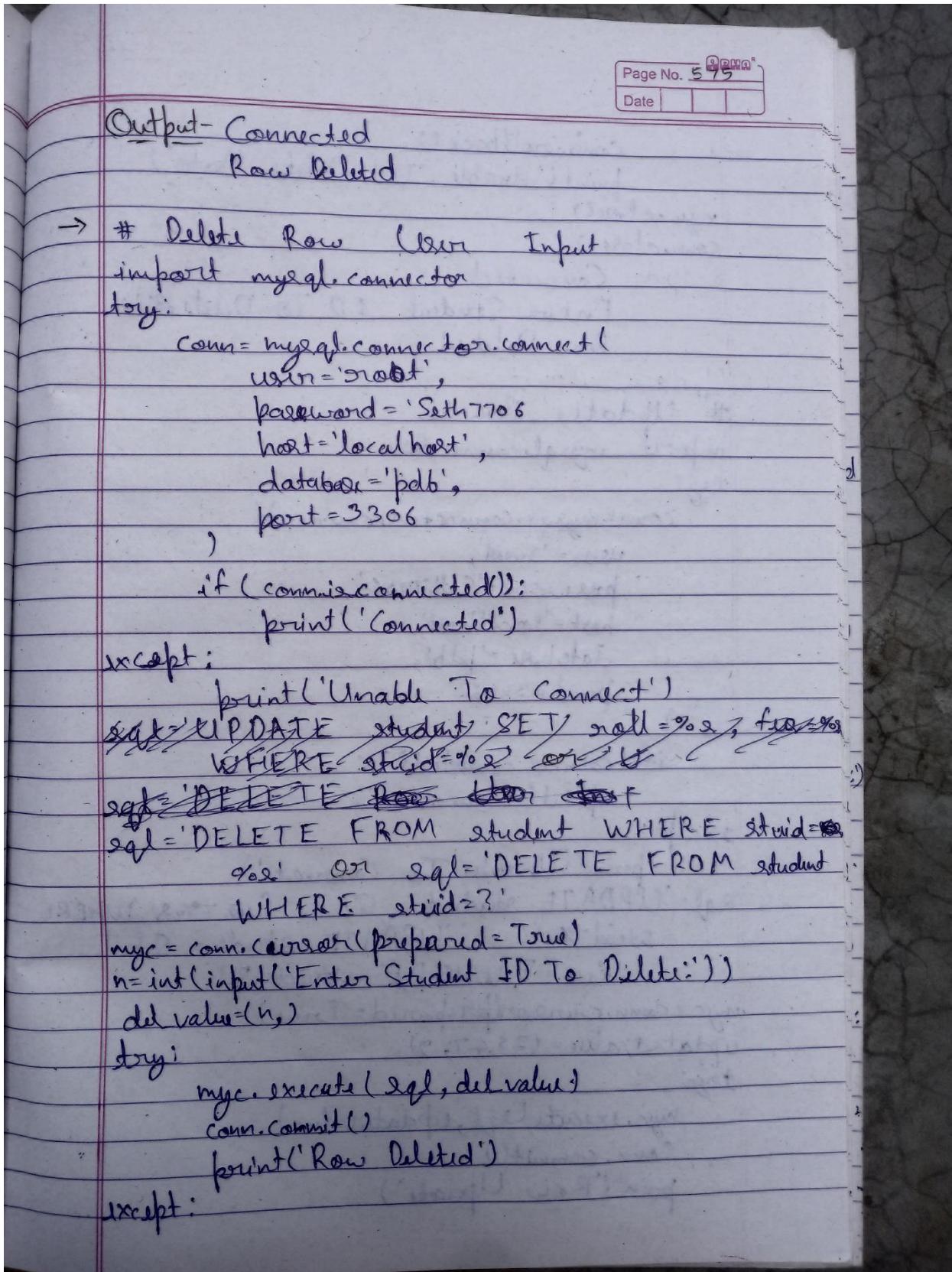
```

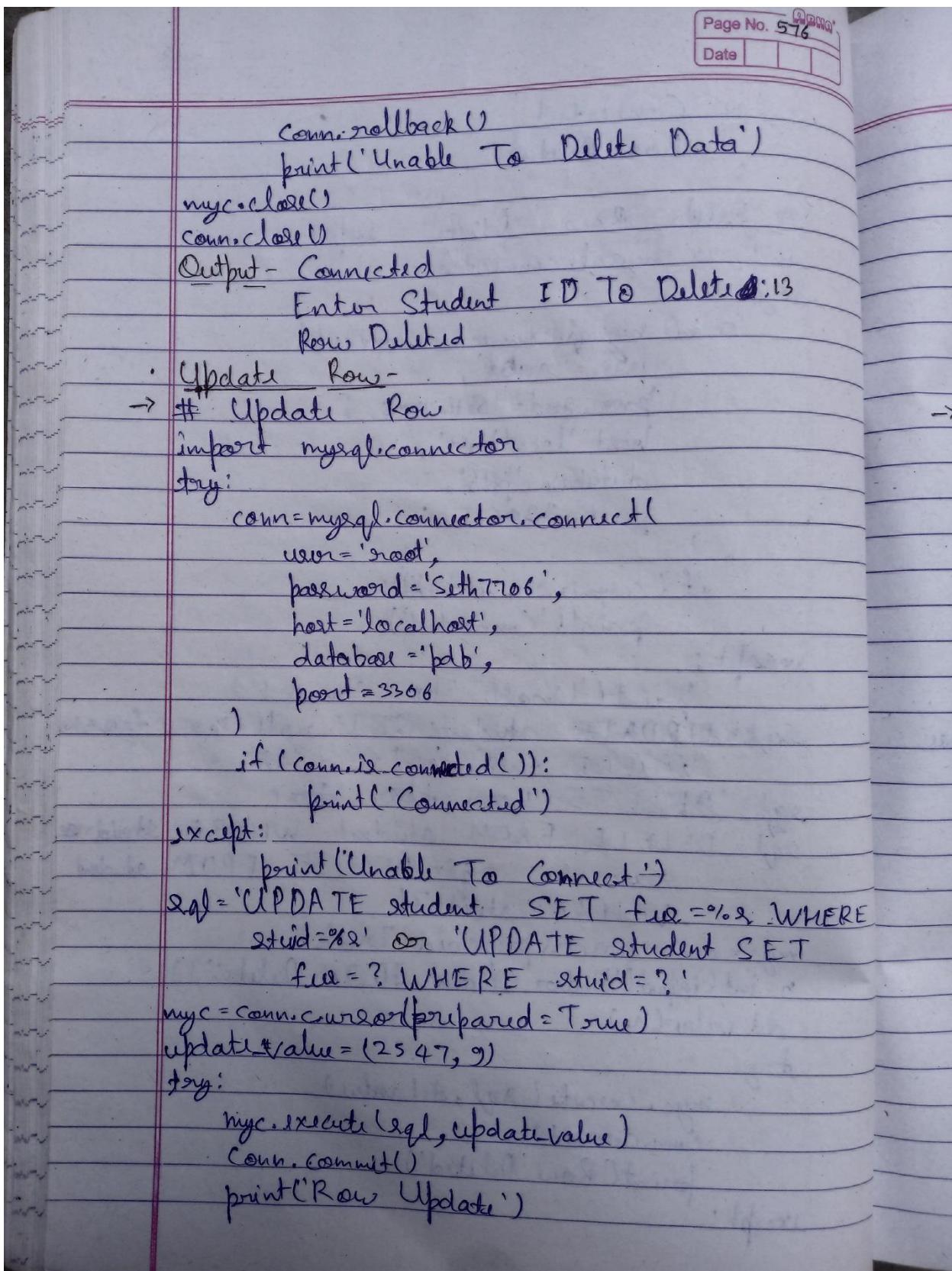












Page No. 57 Date \_\_\_\_\_

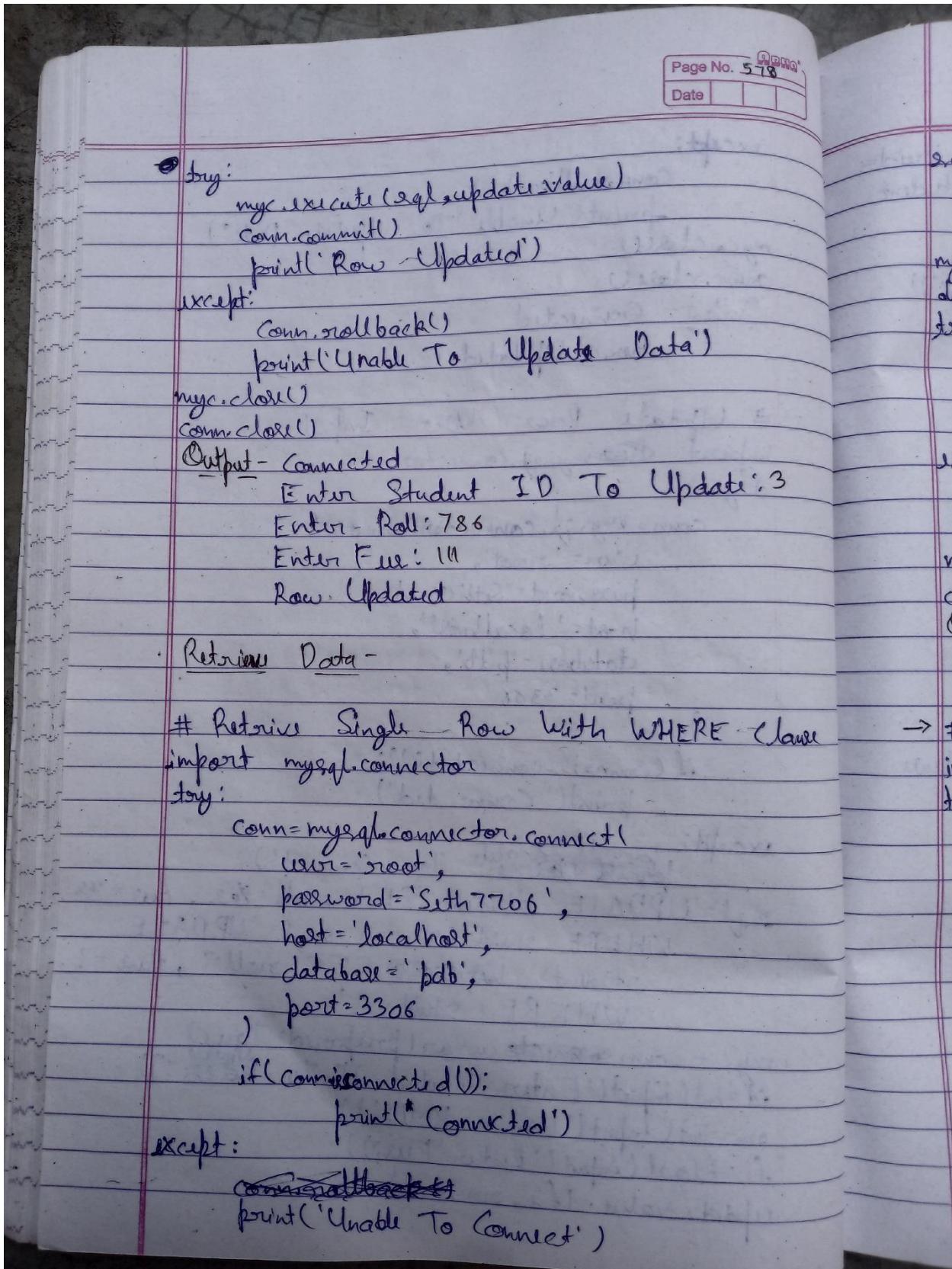
```

except:
    conn.rollback()
    print('Unable To Update Data')
myc.close()
conn.close()

Output - Connected
Row Updated

→ # Update Row User Input
import Remysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if conn.is_connected():
        print('Connected')
except:
    print('Unable To Connect')
sql = 'UPDATE student SET roll=%s, fee=%s WHERE studid=%s' or sql = 'UPDATE
student WHERE SET roll=? , fee=?'
WHERE studid = ?'
myc = conn.cursor(prepared=True)
id = int(input('Enter Student ID To Update :'))
no = int(input('Enter Roll: '))
fe = float(input('Enter Fee: '))
update_value = (fe, no, id)

```



Page No. 579  
Date \_\_\_\_\_

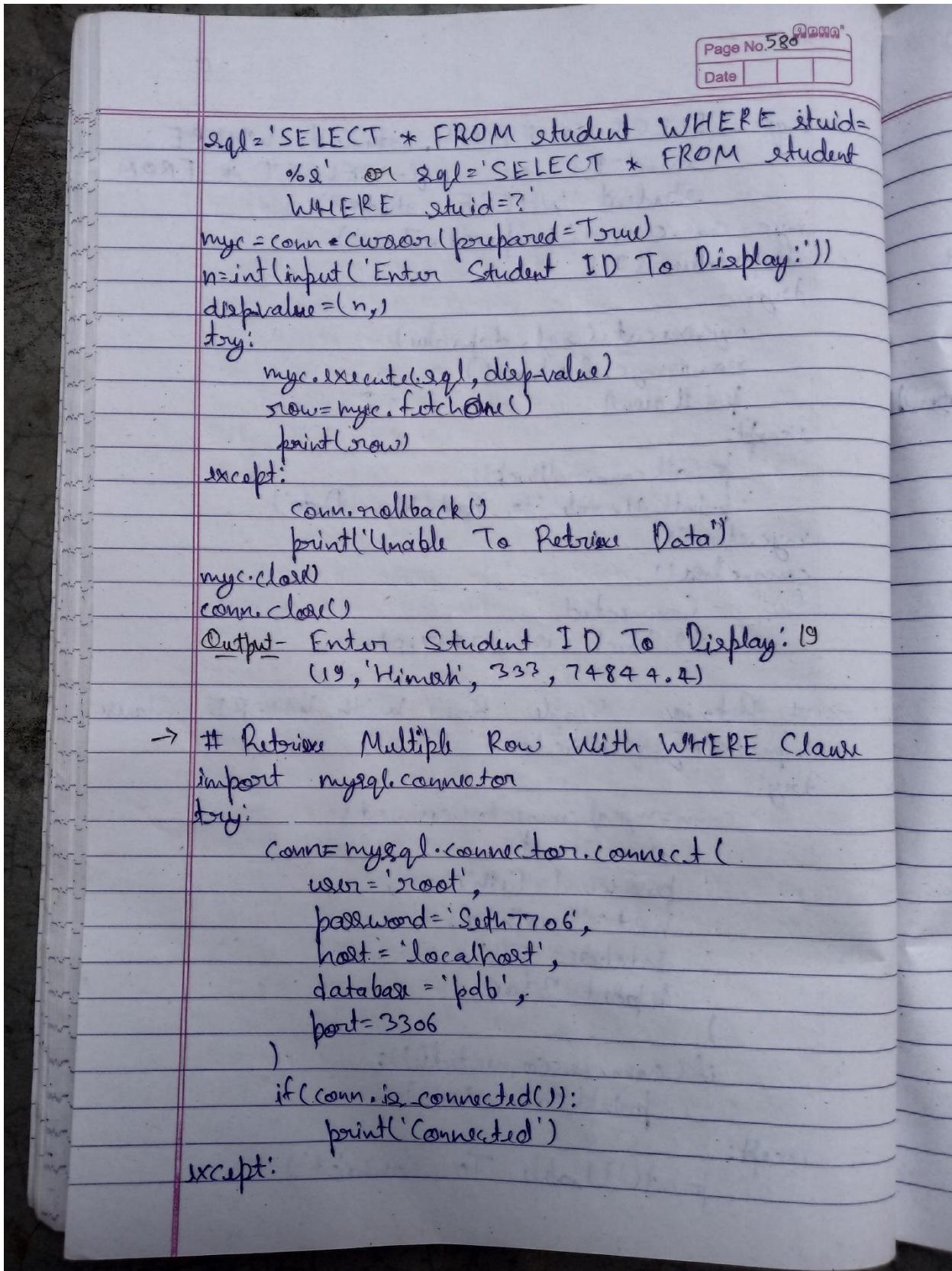
```

sql = 'SELECT * FROM student WHERE
       stud_id=%s' or sql = 'SELECT * FROM
       student WHERE stud_id=?'
myc = conn.cursor(prepared=True)
disp_value = (9,)
try:
    myc.execute(sql, disp_value)
    row = myc.fetchone()
    print(row)
except:
    conn.rollback()
    print('Unable To Retrieve Data')
myc.close()
conn.close()

Output - Connected
(9, 'Jack', 125, 2547.0)

→ # Retrieve Single Row With WHERE Clause
import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )
    if (conn.is_connected()):
        print('Connected')
except:
    print('Unable To Connect')

```



Page No. 561  
Date

```

print('Unable To Connect')
sql = 'SELECT * FROM student WHERE fee fee=500'. or sql = 'SELECT * FROM
student WHERE fee=500'
myc = conn.cursor(prepared=True)
disp_value = (5000,)
try:
    myc.execute(sql, disp_value)
    row = myc.fetchone()
    while row is not None:
        print(row)
        row = myc.fetchone()
except:
    conn.rollback()
    print('Unable To Retrieve Data')
myc.close()
conn.close()

Output - Connected
(4, 'Veenu', 104, 5000.0)
(16, 'Sumit', 152, 5000.0)

→ # Retrieve Multiple Row With WHERE Clause User
Input
import mysql.connector
try:
    conn = mysql.connector.connect(
        user='root',
        password='Seth7706',
        host='localhost',
        database='pdb',
        port=3306
    )

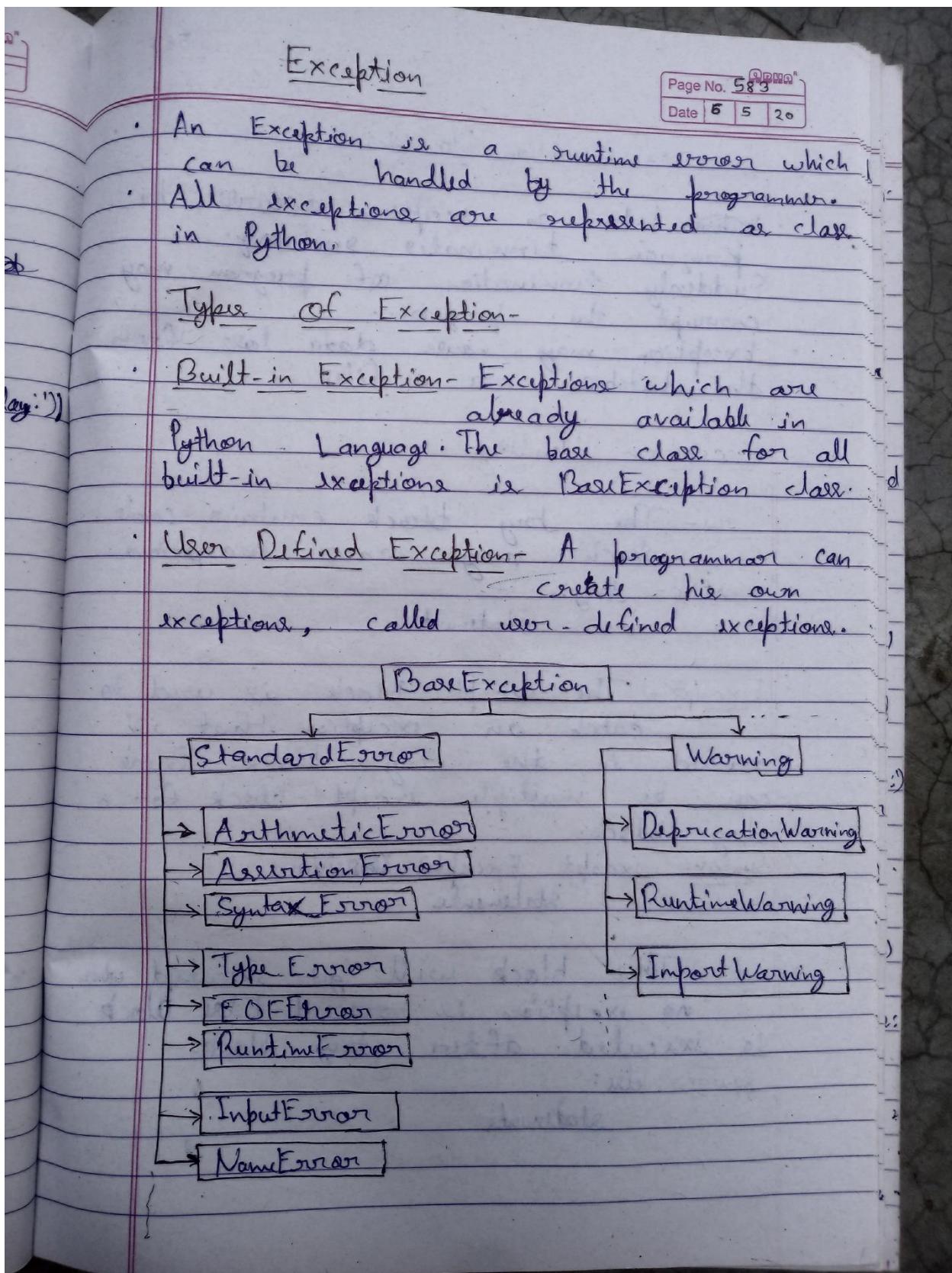
```

Page No. 532 Date \_\_\_\_\_

```

if (conn.isconnected()):
    print('Connected')
except:
    print('Unable To Connect')
sql='SELECT * FROM student WHERE fees
    fees=%s' or sql='SELECT * FROM
    student WHERE fees=?'
myc=conn.cursor(just_pseudo=True)
fe= float(input('Enter Student Fee To Display'))
disp_value(fe)
try:
    myc.execute(sql, disp_value)
    row=myc.fetchone()
    while row is not None:
        print(row)
        row=myc.fetchone()
except:
    conn.rollback()
    print('Unable To Retrive Data')
myc.close()
conn.close()
Output- Connected
Enter student Fee To Display:5000
(4, 'Verni', 104, 5000.0)
(16, 'Sumit', 152, 5000.0)

```



Page No. 584  
Date

### Need of Exception Handling -

- When an exception occurs, the program terminates suddenly.
- Suddenly termination of program may corrupt the program.
- Exception may cause data loss from the database or a file.

### Exception Handling -

- Try - The try block contains code which may cause exceptions.

Syntax - try:  
statements

- Except - The except block is used to catch an exception that is raised in the try block. There can be multiple except block for a try block.

Syntax - except ExceptionClassName  
statements.

- Else - This block will get executed when no exception is raised. Else block is executed after try block.

Syntax - else:  
statements

Page No. 585  
Date

Finally - This block will get executed irrespective of whether there is an exception or not.

Syntax - finally:  
statements.

Rules -

- We can write several except blocks for a single try block.
- We can write multiple except blocks to handle multiple exceptions.
- We can write try block without any except blocks.
- We ~~can~~ can not write except block without a try block.
- Finally block is always executed ~~is~~ irrespective of whether there is an exception or not.
- Else block is optional.
- Finally block is optional.

Syntax - try:  
statements  
except ExceptionClassName:  
statements  
else:  
statements  
finally:  
statements.

Page No. 586

Date

Syntax 2 - try:

    statements  
except ExceptionClassName:  
    statements

Syntax 3 - try:

    statements  
except ExceptionClassName:  
    statements  
except ExceptionClassName:  
    statements  
finally:  
    statements.

Syntax 4 - try:

    statements

Except -

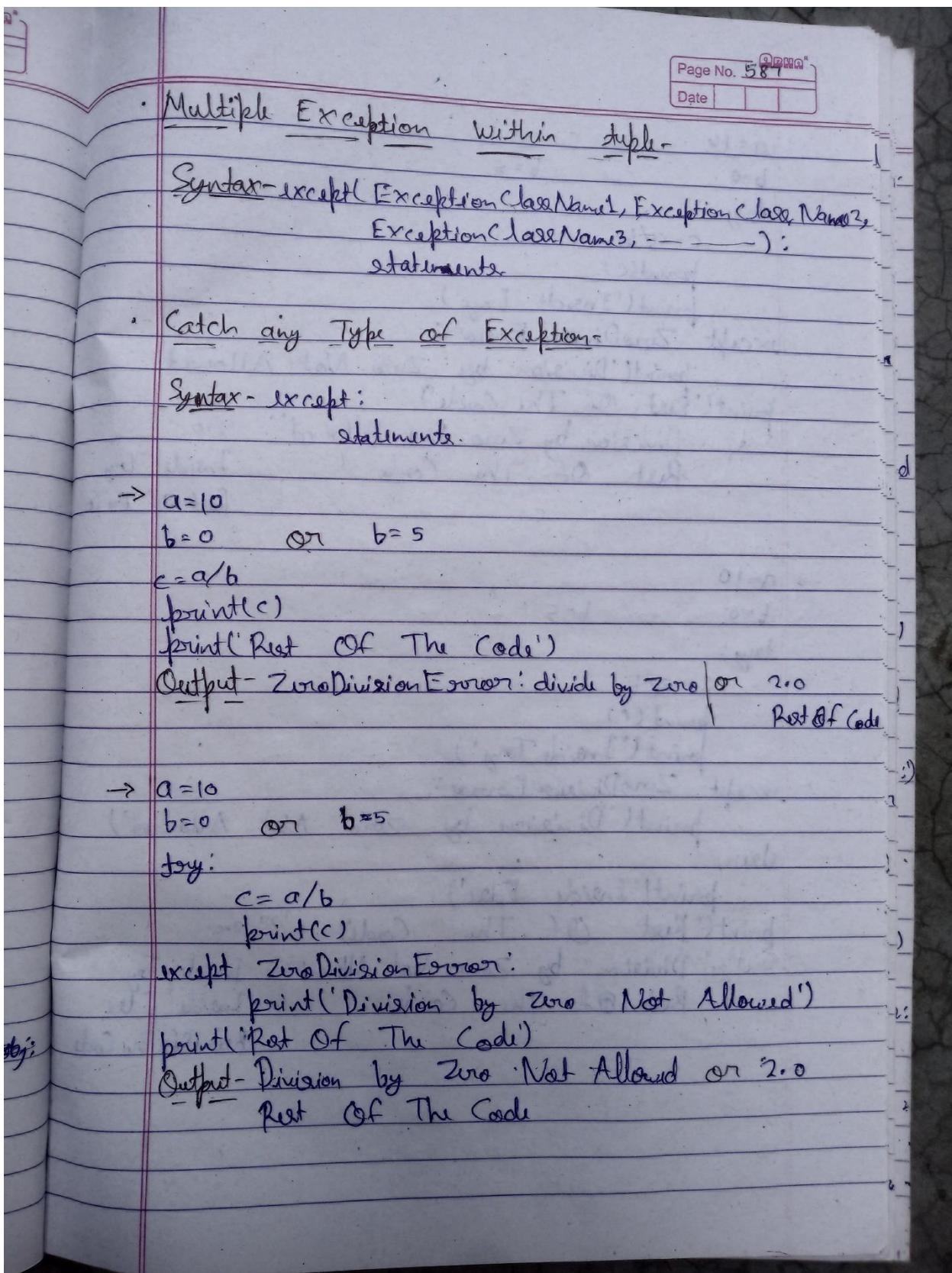
- With the Exception Class Name -

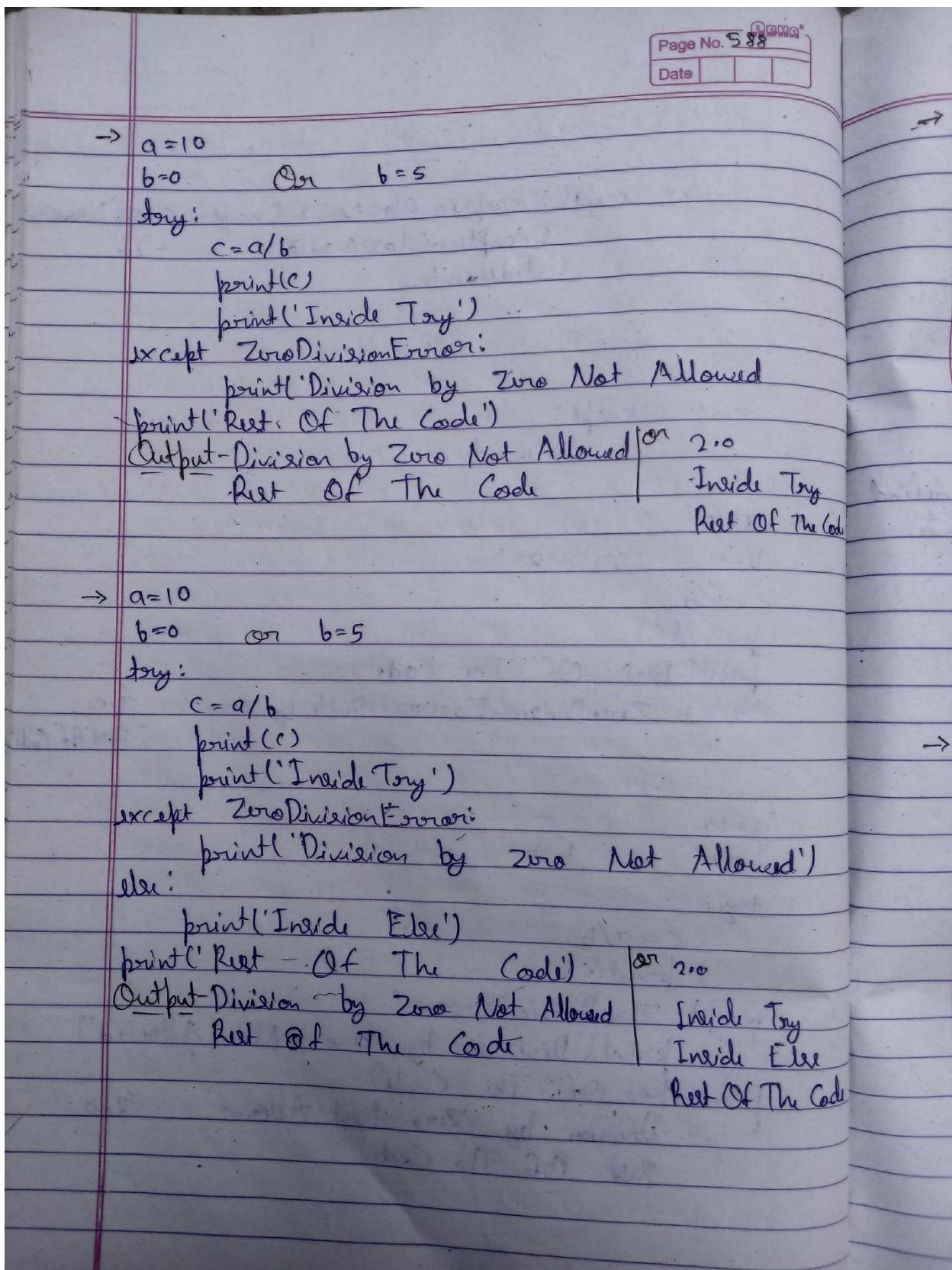
Syntax - except Exception Class Name:  
    statements

- ~~Exception~~ as - an object -

~~Syntax except~~ ~~Exception~~ ~~Exception Class Name as obj~~  
~~statements~~

Syntax - except





Page No. 589  
Date \_\_\_\_\_

→  $a=10$   
 $b=0$       or       $b=5$

```

try:
    c=a/b
    print(c)
except ZeroDivisionError:
    print('Division by Zero Not Allowed')
else:
    print('Inside Else')
finally:
    print('Inside Finally')
print('Rest Of The Code')

```

Output-Division by Zero Not Allowed or 2.0

Inside Finally	Inside Else
Rest Of The Code	Inside Finally
Rest Of The Code	Rest Of The Code

→  $a=10$   
 $b=0$

```

try:
    c=4/b
    print(c)
except:
    print('Inside Try')
    print('ZeroDivisionError as obj')
    print(obj)
print('Rest Of The Code')

```

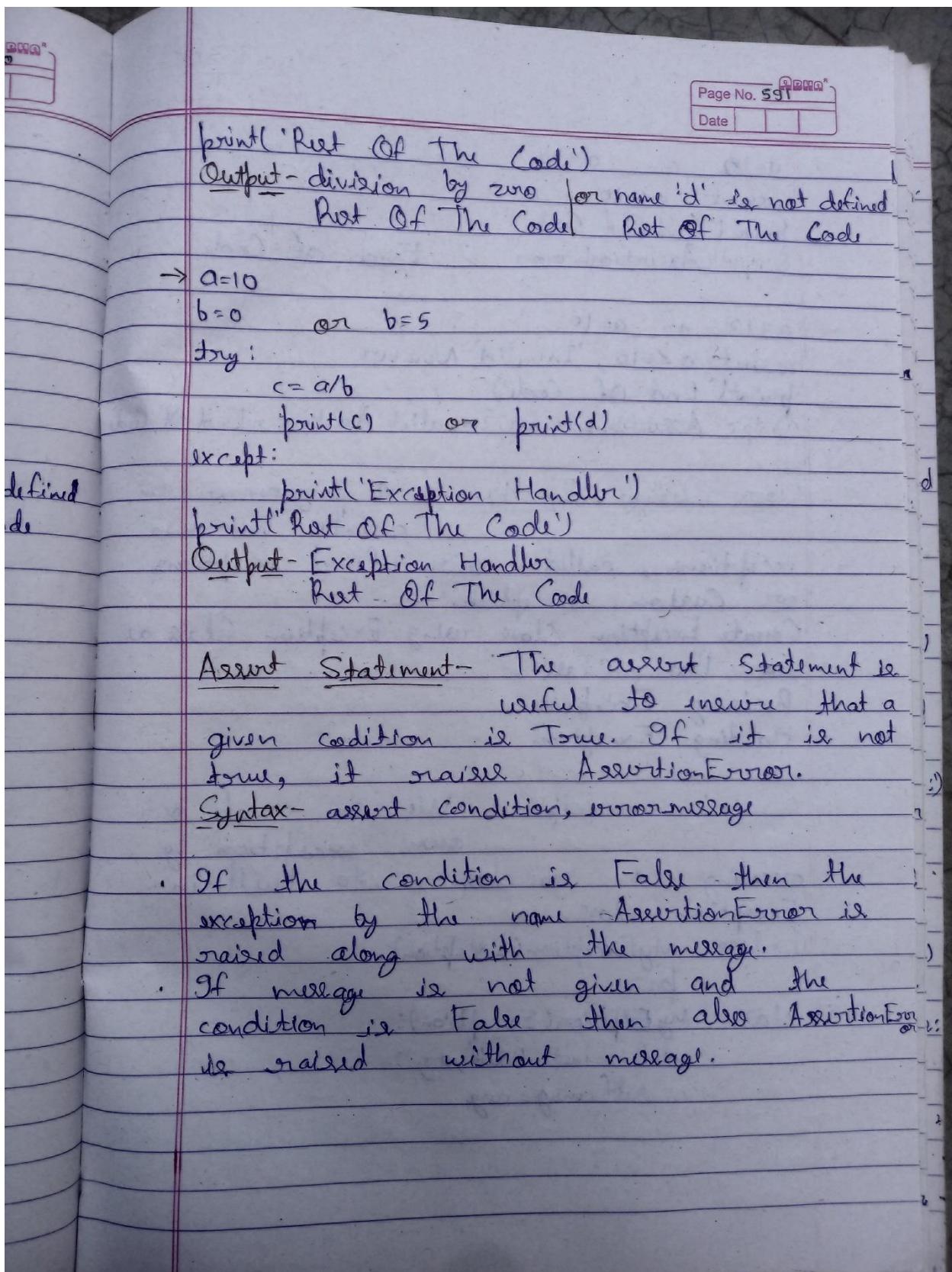
Output-division by zero or 2.0

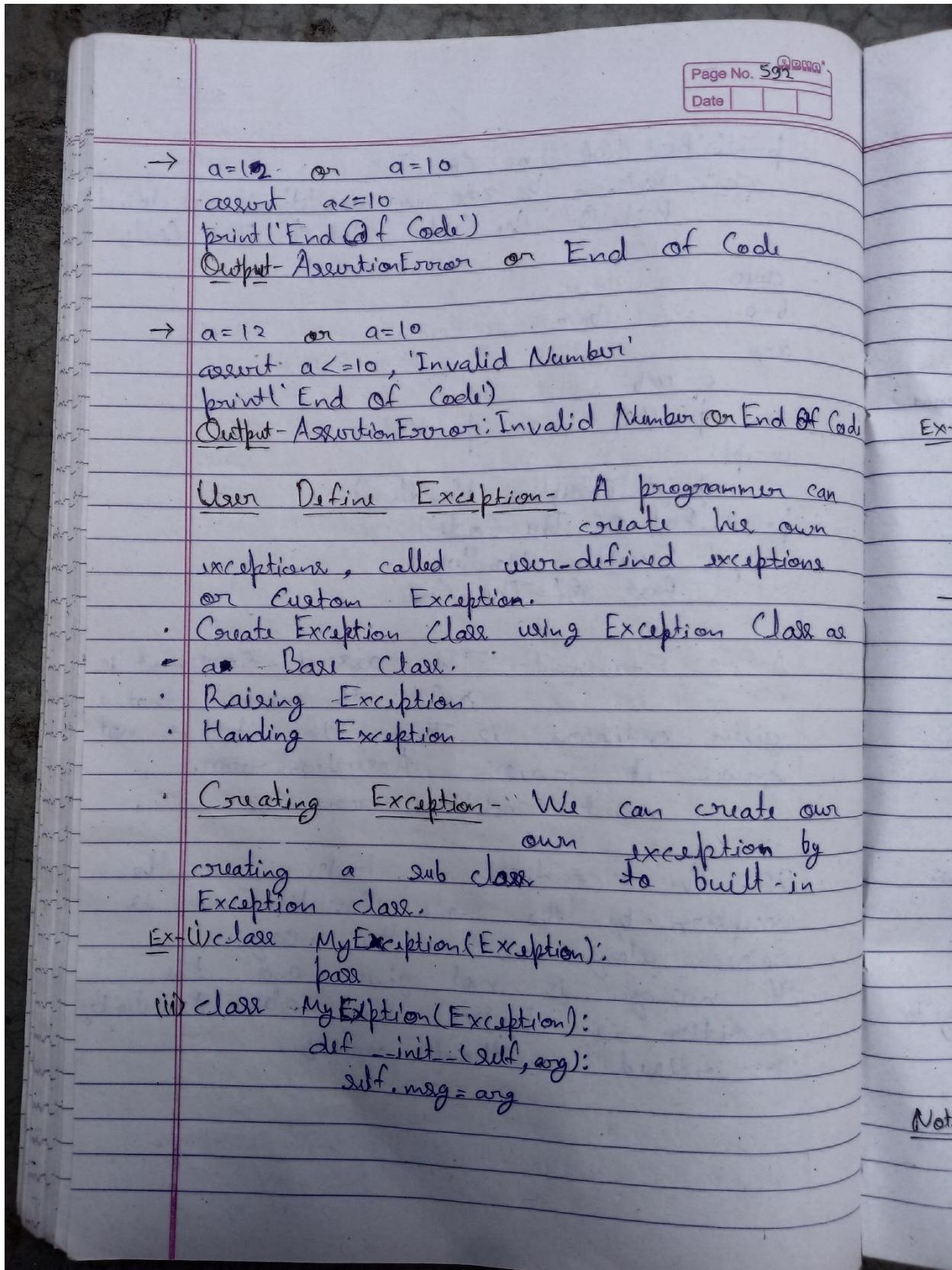
Rest Of The Code	Inside Try
Rest Of The Code	Rest Of The Code

$\rightarrow a = 10$   
 $b = 0 \text{ or } b = 5$   
**try:**  
 $c = a/b$   
 $\text{print}(c) \text{ or } \text{print}(d)$   
**except ZeroDivisionError as obj:**  
 $\text{print}(obj)$   
**except NameError as obj:**  
 $\text{print}(obj)$   
**print('Rest Of The Code')**  
Output- division by zero | on name 'd' is not defined  
 Rest Of The Code | Rest Of The Code

$\rightarrow a = 10$   
 $b = 0$   
**try:**  
 $c = a/b$   
 $\text{print}(c)$   
**except (NameError, ZeroDivisionError):**  
 $\text{print}(\text{Exception Handler})$   
**print('Rest Of The Code')**  
Output- Exception Handler  
 Rest Of The Code

$\rightarrow a = 10$   
 $b = 0 \text{ or } b = 5$   
**try:**  
 $c = a/b$   
 $\text{print}(c) \text{ or } \text{print}(d)$   
**except (NameError, ZeroDivisionError) as obj:**  
 $\text{print}(obj)$





Page No. 593 Date

- Raising Exception - raise statement is used to raise the user defined exception.
- Syntax - raise MyException('message')
- Handling Exception - Using try and except block programmer can handle exceptions.

Code

```

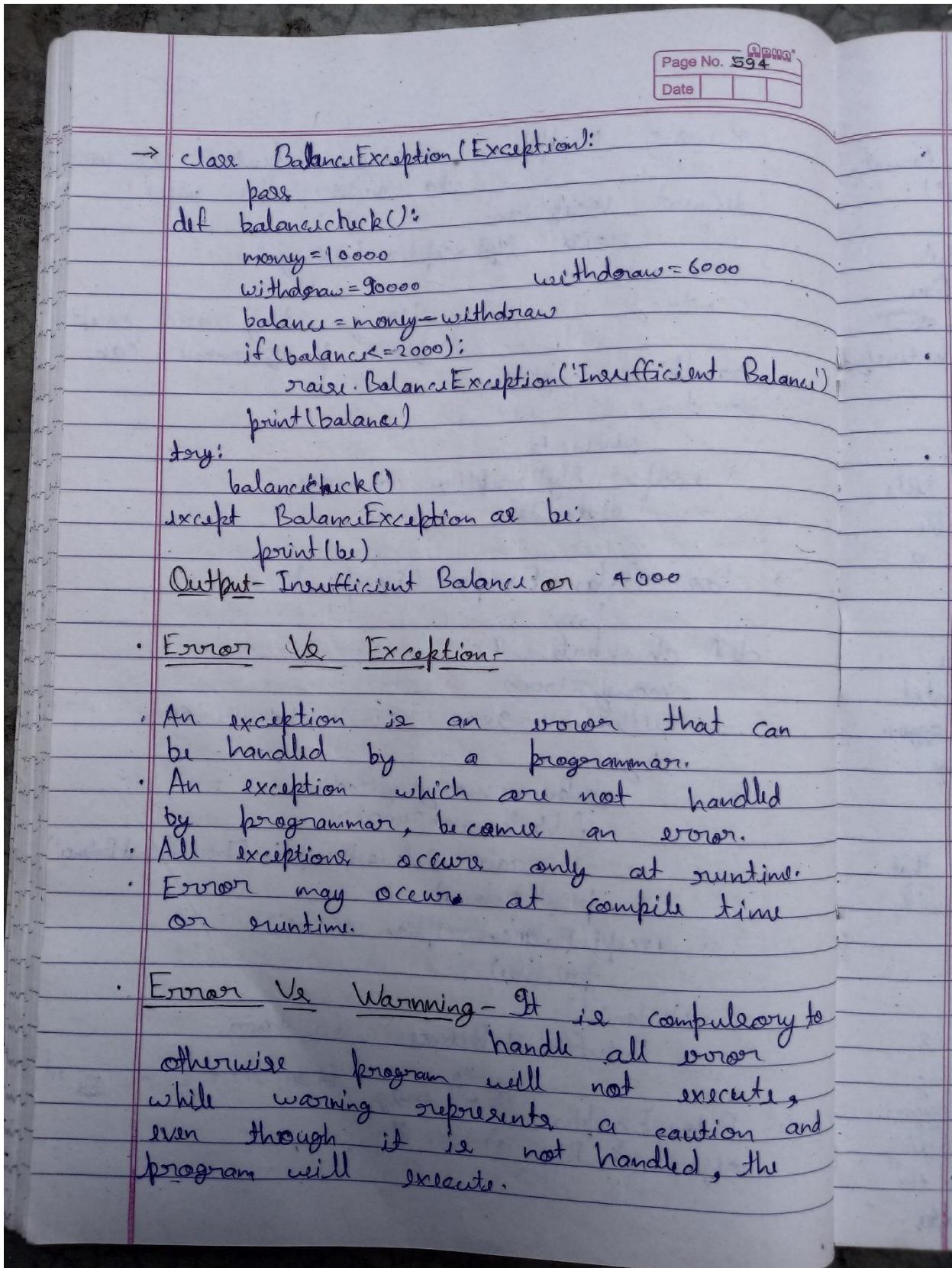
Ex- try:
      statements
except MyException as my:
      statements
  
```

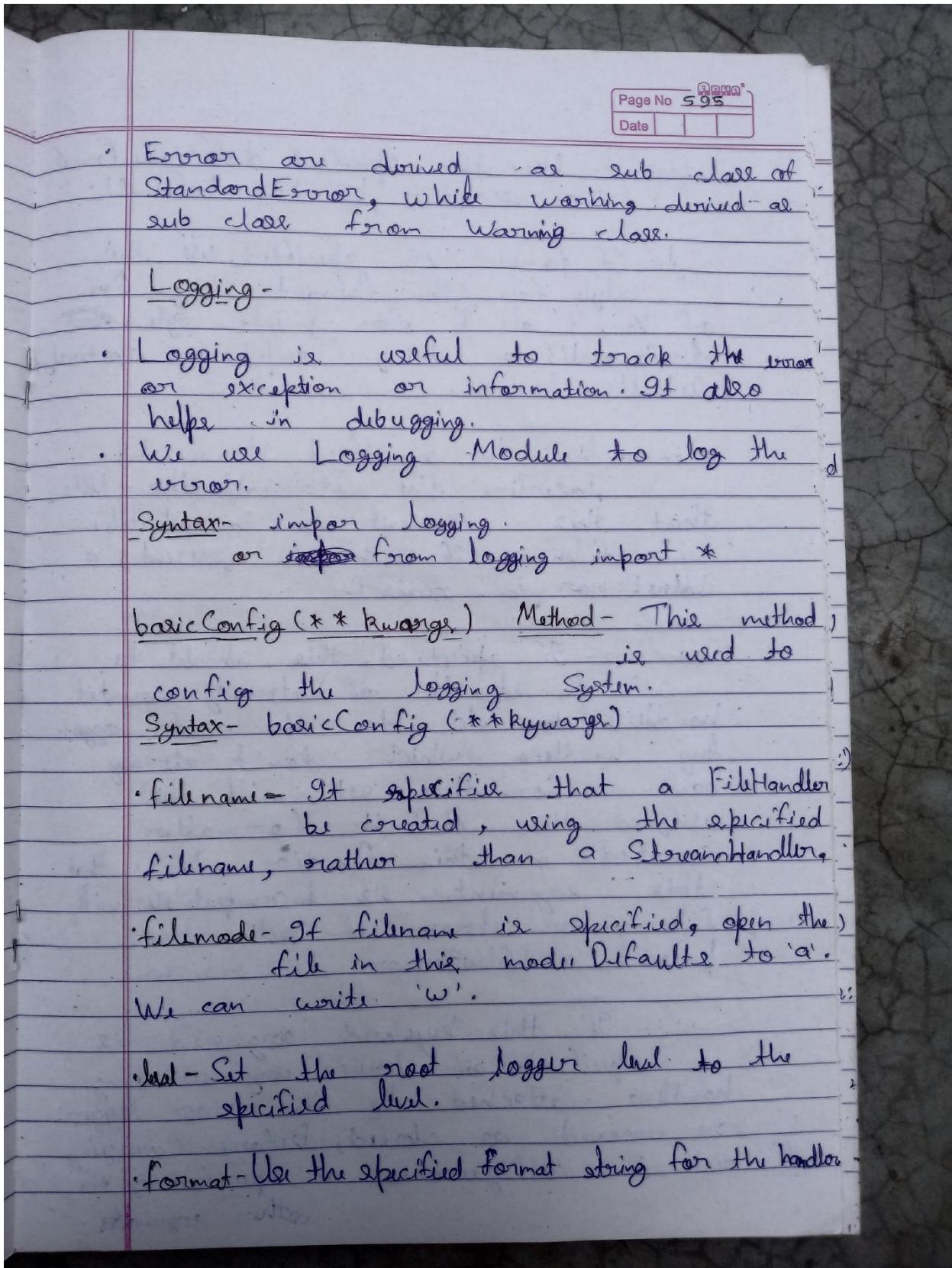
```

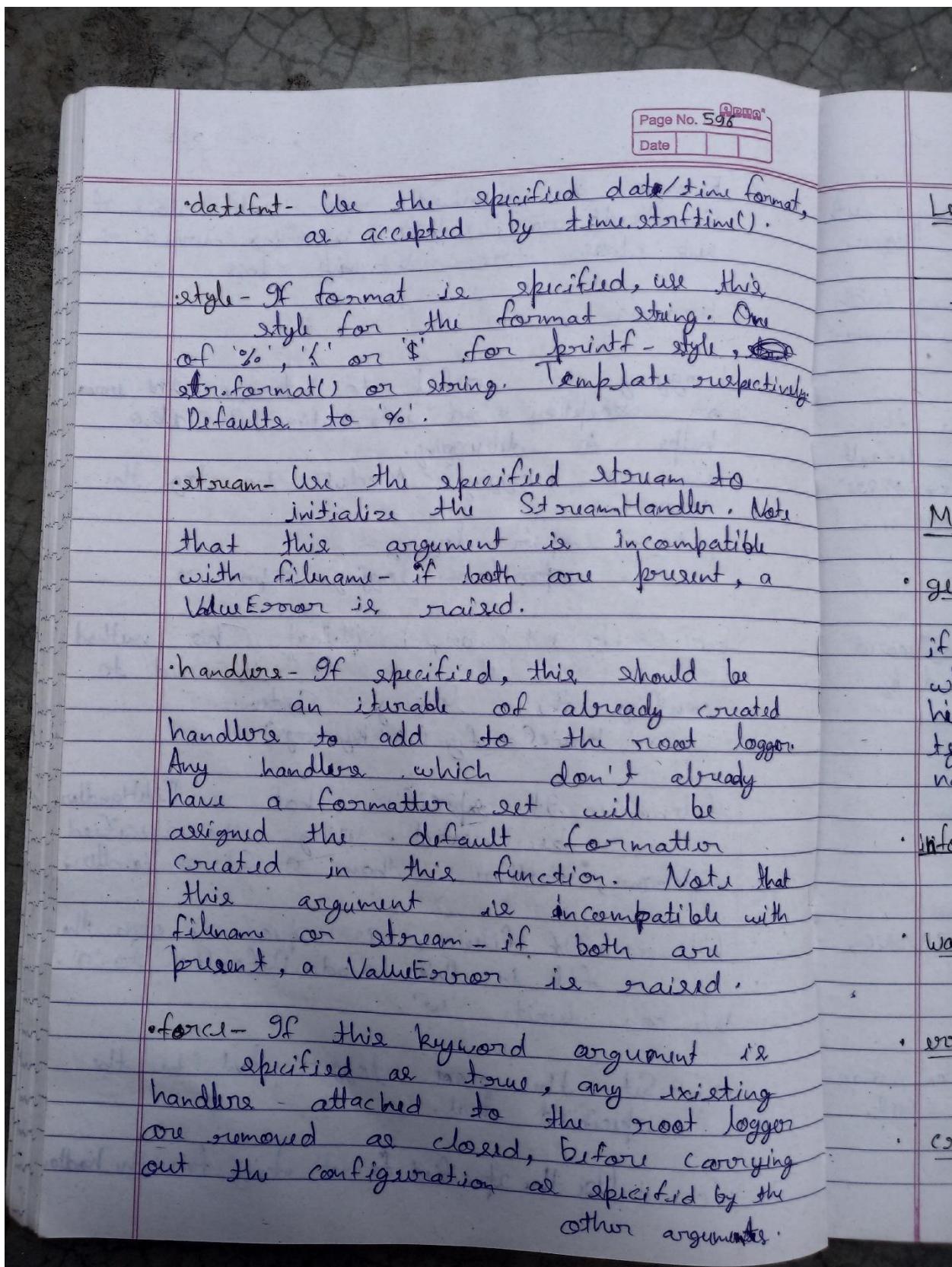
→ class BalanceException(Exception):
    pass
    def checkbalance():
        money = 10000
        withdraw = 9000 or withdraw = 6000
        try:
            balance = money - withdraw
            if (balance <= 2000):
                raise BalanceException('Insufficient Balance')
                print(balance)
            except BalanceException as be:
                print(be)
        checkbalance()
    
```

Output - Insufficient Balance or 4000

Note - BalanceException फ़िर मैंग तो 2011 में ये दे गया था  
Object नहीं Pass होता है।







Page No. 597 Date \_\_\_\_\_

Levels - (Default Numerical Value is 30.)

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Methods -

- getLogger() - This method returns a logger with the specified name or, if name is None, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like 'a', 'a.b' or 'a.b.c.d'.
- info(msg) - This will log a message with level INFO on this logger.
- warning(msg) - This will log a message with level WARNING on this logger.
- error(msg) - This will log a message with level ERROR on this logger.
- critical(msg) - This will log a message with level CRITICAL on this logger.

