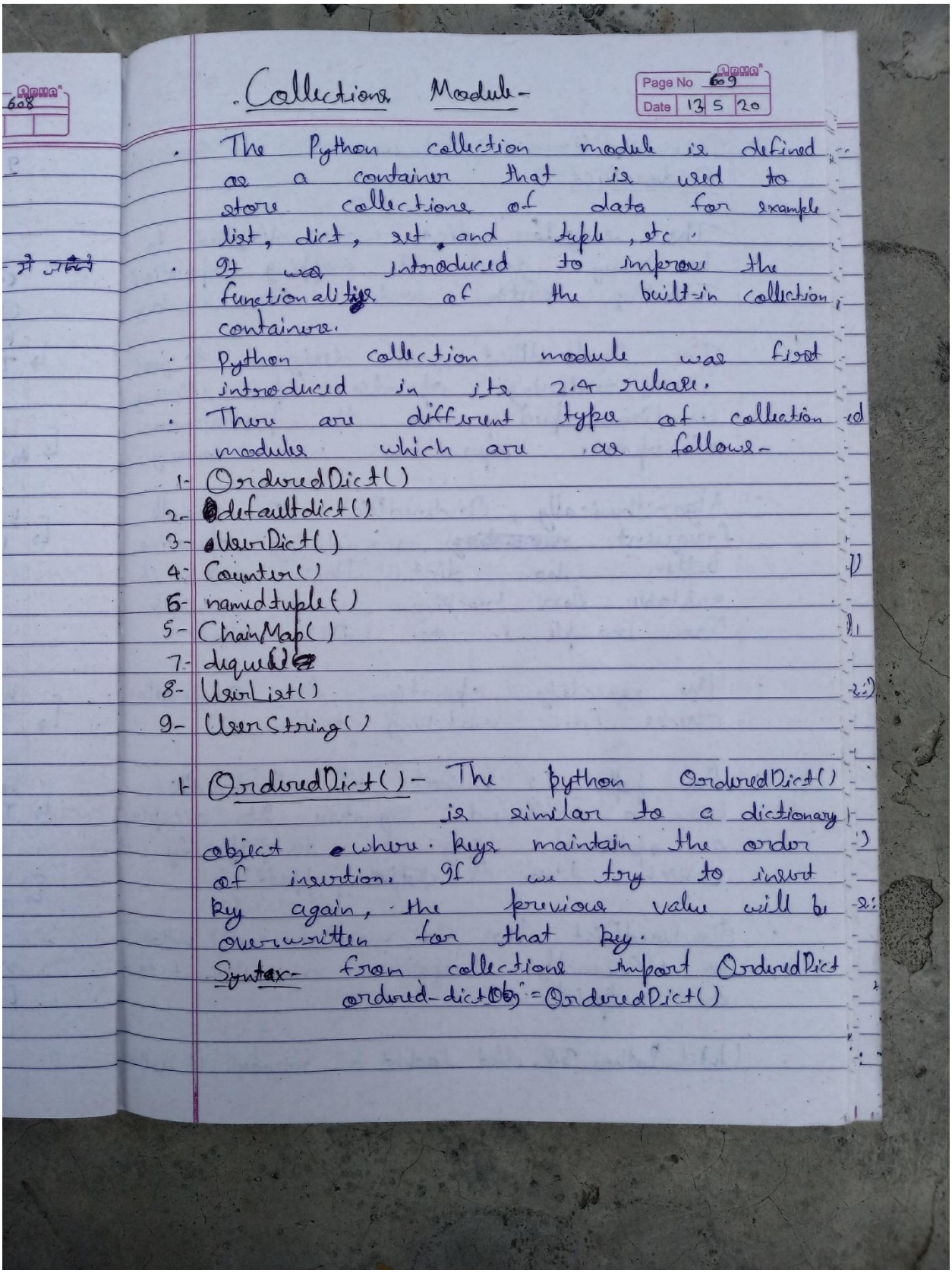
Python Collections Module

Source Code- https://github.com/satyam-seth-learnings/python_learning/tree/main/Python%20Built-in%20Modules/Python%20Collections%20Module

SATYAM SETH

22-09-2021



Collections Module-

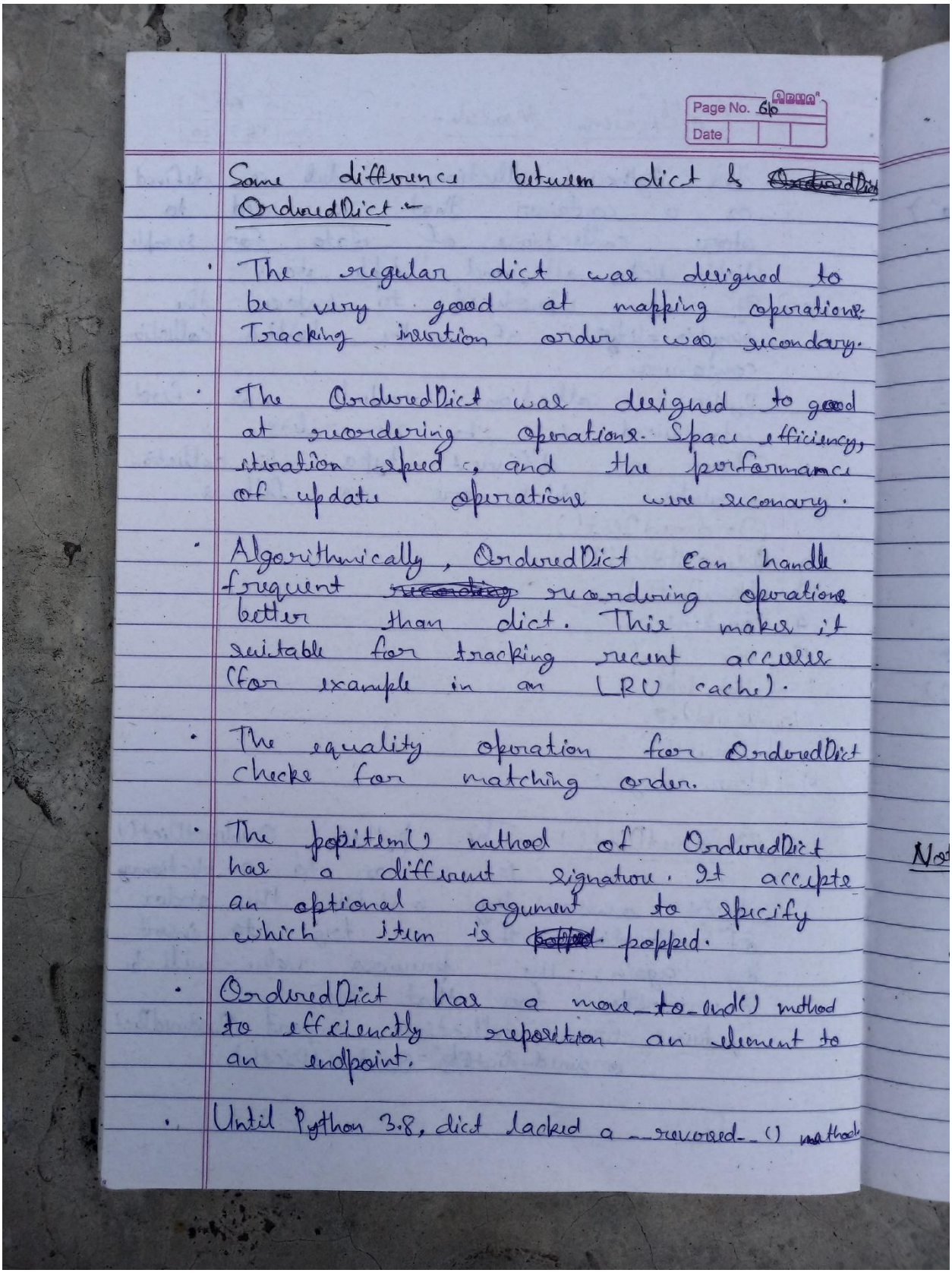
Page No 609
Date 13/5/20

- The Python collection module is defined as a container that is used to store collections of data for example list, dict, set, and tuple, etc.
- It was introduced to improve the functionalities of the built-in collection containers.
- Python collection module was first introduced in its 2.4 release.
- There are different type of collection module which are as follows-

1- OrderedDict()
2- defaultdict()
3- UserDict()
4- Counter()
5- namedtuple()
5- ChainMap()
7- deque()
8- UserList()
9- UserString()

1- **OrderedDict()**- The python OrderedDict() is similar to a dictionary object where keys maintain the order of insertion. If we try to insert key again, the previous value will be overwritten for that key.

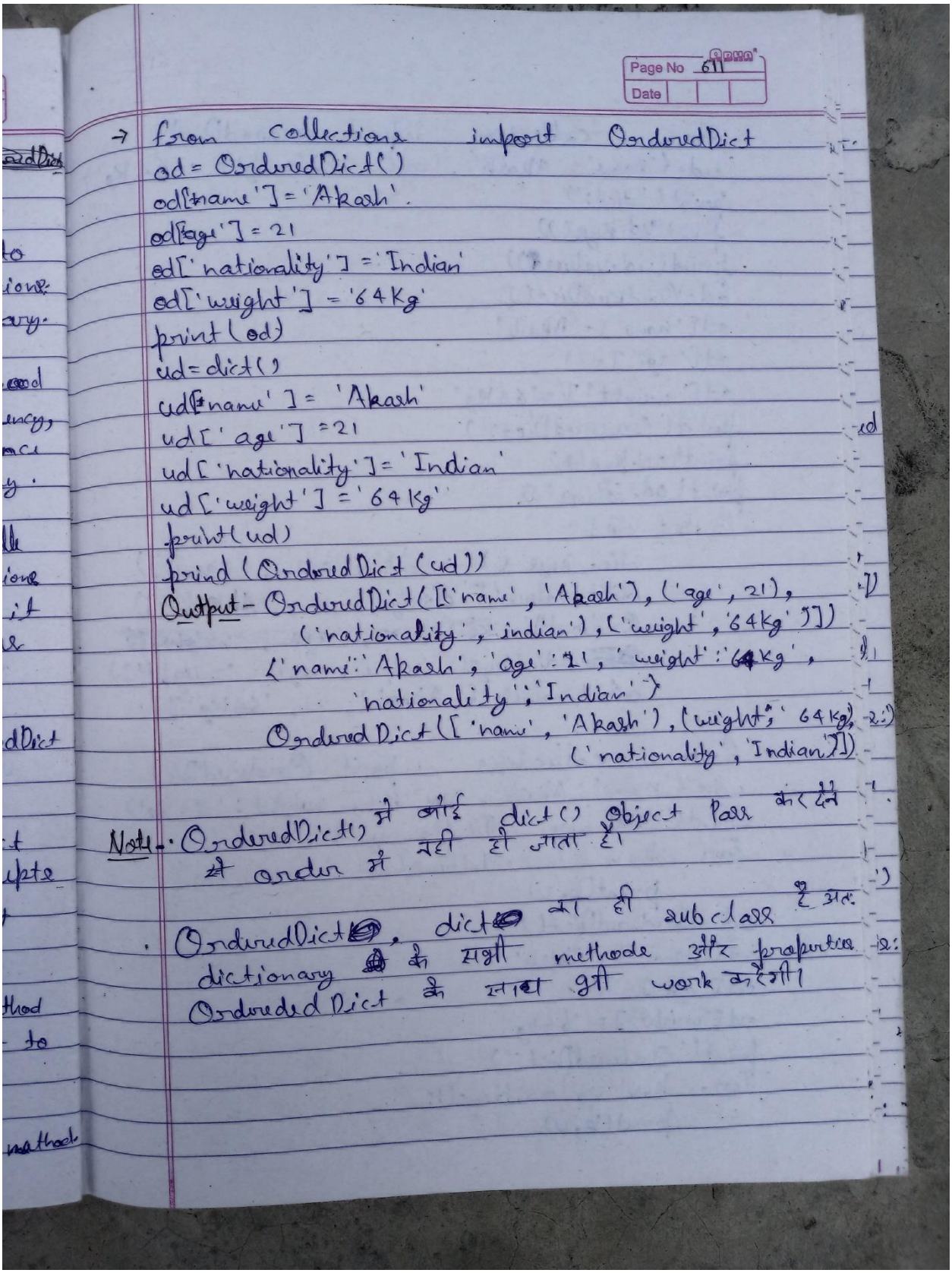
Syntax- from collections import OrderedDict
ordered_dict_Obj = OrderedDict()



Page No. 66
Date

Some difference between dict & ~~OrderedDict~~ OrderedDict :-

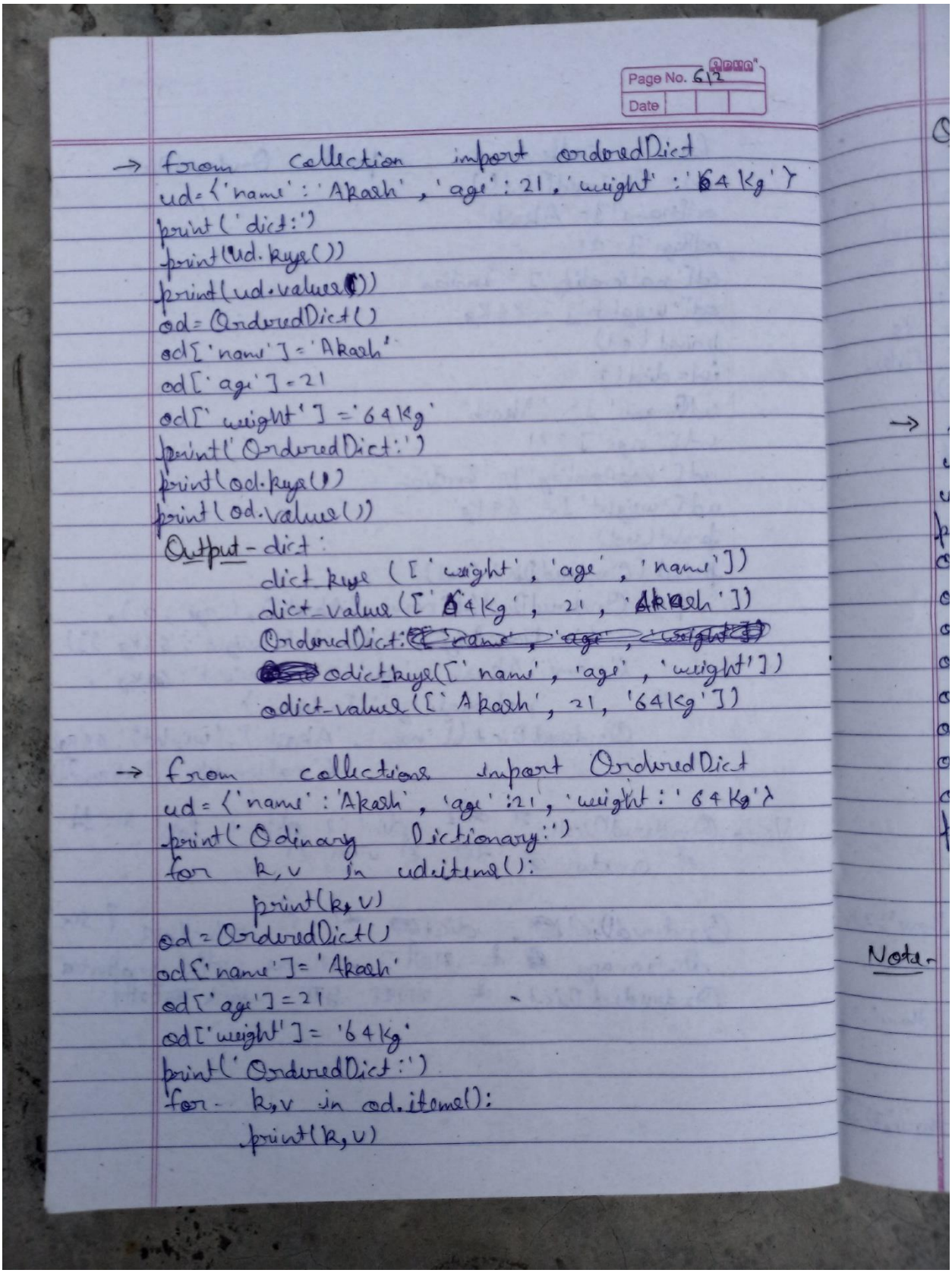
- The regular dict was designed to be very good at mapping operations. Tracking insertion order was secondary.
- The OrderedDict was designed to good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- Algorithmically, OrderedDict can handle frequent ~~reordering~~ reordering operations better than dict. This makes it suitable for tracking recent accesses (for example in an LRU cache).
- The equality operation for OrderedDict checks for matching order.
- The popitem() method of OrderedDict has a different signature. It accepts an optional argument to specify which item is ~~popped~~ popped.
- OrderedDict has a move_to_end() method to efficiently reposition an element to an endpoint.
- Until Python 3.8, dict lacked a __reversed__() method.



→ from collections import OrderedDict
od = OrderedDict()
od['name'] = 'Akash'
od['age'] = 21
od['nationality'] = 'Indian'
od['weight'] = '64Kg'
print(od)
ud = dict()
ud['name'] = 'Akash'
ud['age'] = 21
ud['nationality'] = 'Indian'
ud['weight'] = '64Kg'
print(ud)
print(OrderedDict(ud))
Output - OrderedDict([('name', 'Akash'), ('age', 21), ('nationality', 'indian'), ('weight', '64kg')])
{'name': 'Akash', 'age': 21, 'weight': '64Kg', 'nationality': 'Indian'}
OrderedDict([('name', 'Akash'), ('weight', '64Kg'), ('nationality', 'Indian')])

Note - • OrderedDict() में कोई dict() object pass कर देने से order में नहीं हो जाता है।

• OrderedDict, dict का ही subclass है अतः dictionary के सभी methods और properties OrderedDict के साथ भी work करेंगी।

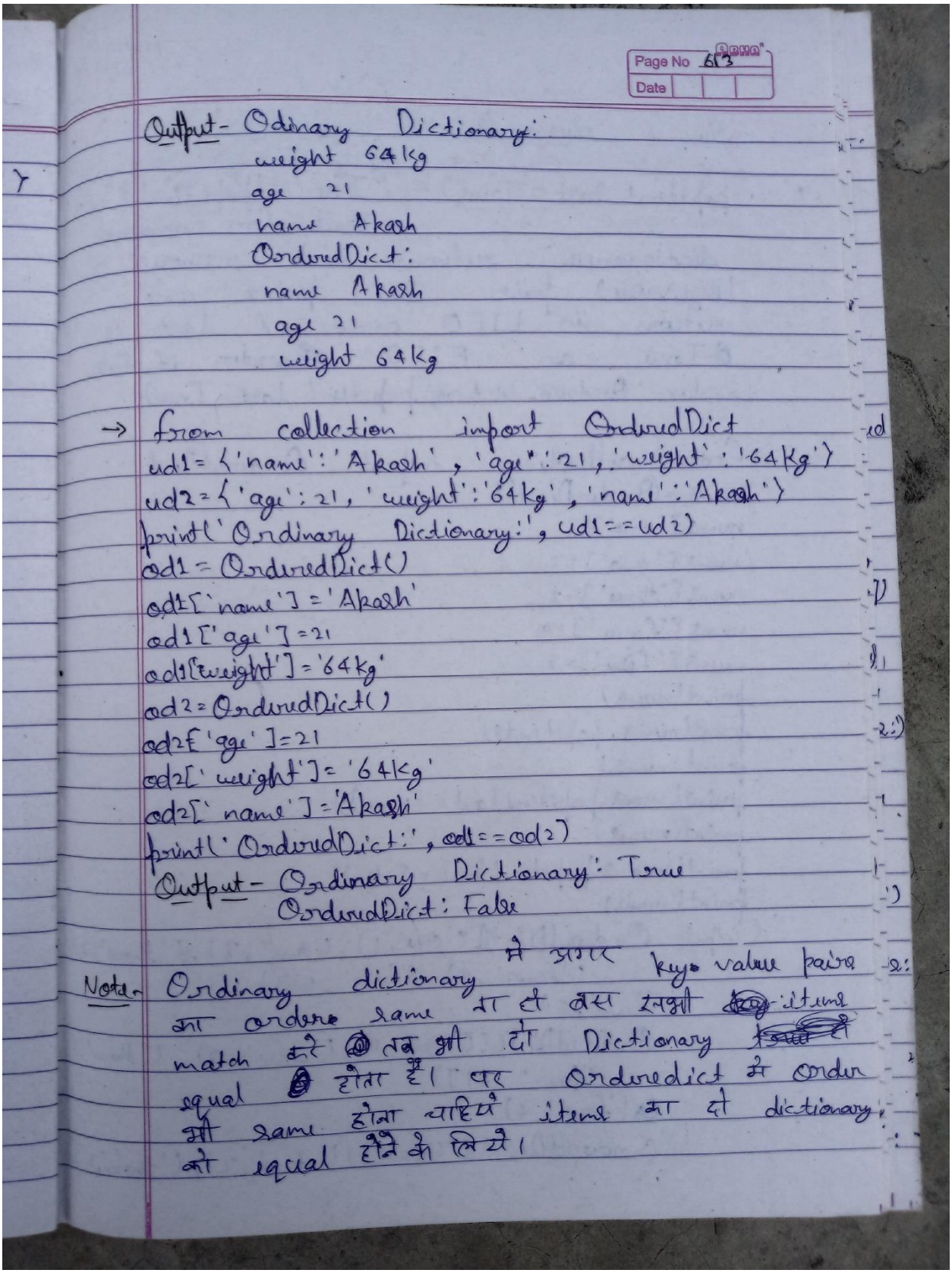


→ from collection import orderedDict
ud={'name':'Akash', 'age':21, 'weight':'64 kg'}
print('dict:')
print(ud.keys())
print(ud.values())
od=OrderedDict()
od['name']='Akash'
od['age']=21
od['weight']='64kg'
print('OrderedDict:')
print(od.keys())
print(od.values())
Output - dict:
dict keys(['weight', 'age', 'name'])
dict_values(['64kg', 21, 'Akash'])
OrderedDict:
odict_keys(['name', 'age', 'weight'])
odict_values(['Akash', 21, '64kg'])

→ from collections import OrderedDict
ud={'name':'Akash', 'age':21, 'weight':'64kg'}
print('Ordinary Dictionary:')
for k,v in ud.items():
print(k,v)
od=OrderedDict()
od['name']='Akash'
od['age']=21
od['weight']='64kg'
print('OrderedDict:')
for k,v in od.items():
print(k,v)

→

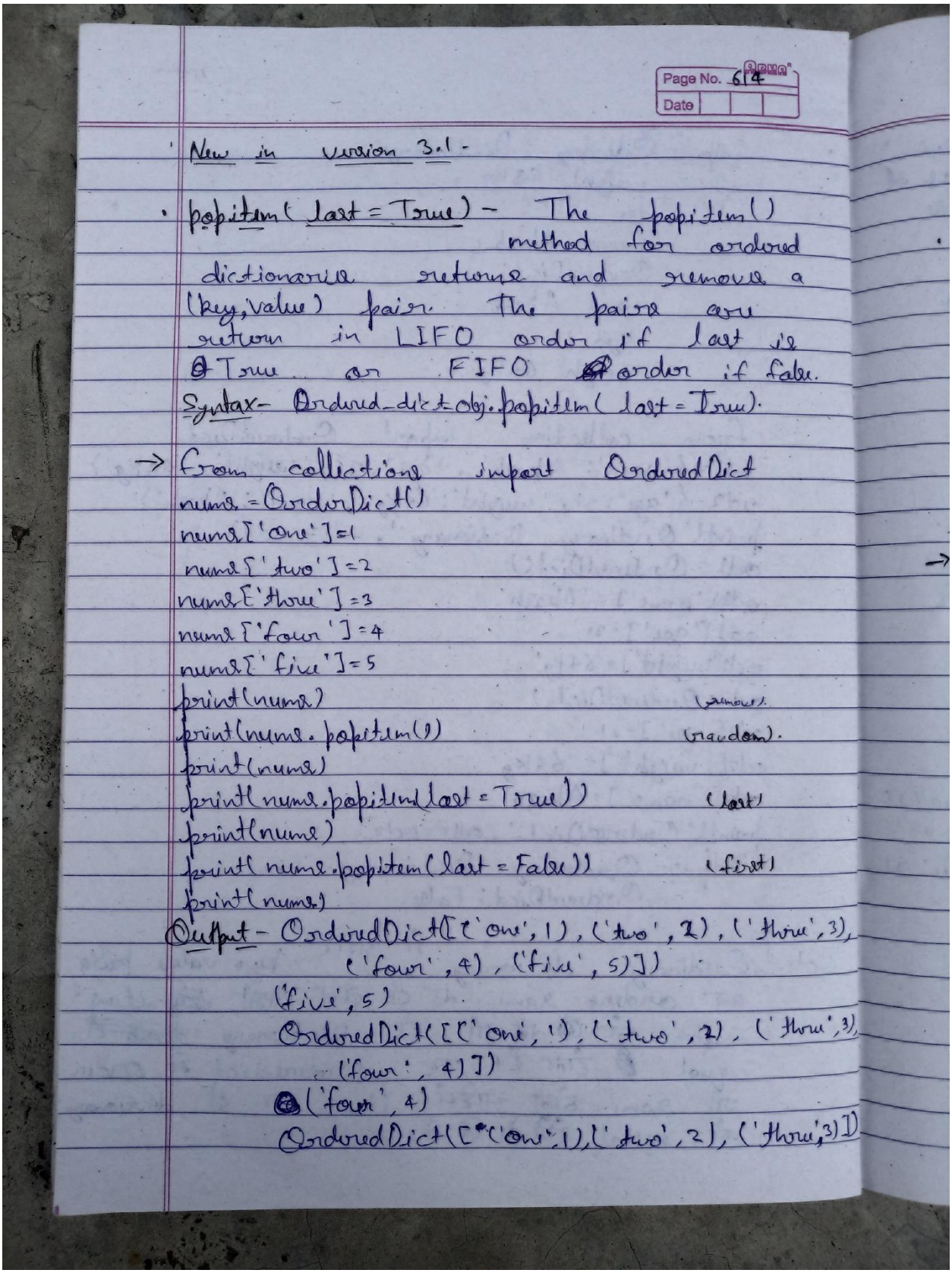
Note-



Output- Ordinary Dictionary:
weight 64kg
age 21
name Akash
OrderedDict:
name Akash
age 21
weight 64kg

→ from collection import OrderedDict
od1 = {'name': 'Akash', 'age': 21, 'weight': '64kg'}
od2 = {'age': 21, 'weight': '64kg', 'name': 'Akash'}
print('Ordinary Dictionary:', od1 == od2)
od1 = OrderedDict()
od1['name'] = 'Akash'
od1['age'] = 21
od1['weight'] = '64kg'
od2 = OrderedDict()
od2['age'] = 21
od2['weight'] = '64kg'
od2['name'] = 'Akash'
print('OrderedDict:', od1 == od2)
Output- Ordinary Dictionary: True
OrderedDict: False

Note- Ordinary dictionary में अगर key-value pairs का order same न हो तब भी दो Dictionary equal होता है। पर Orderedict में order भी same होना चाहिये items का दो dictionary को equal होने के लिये।



Page No. 614
Date

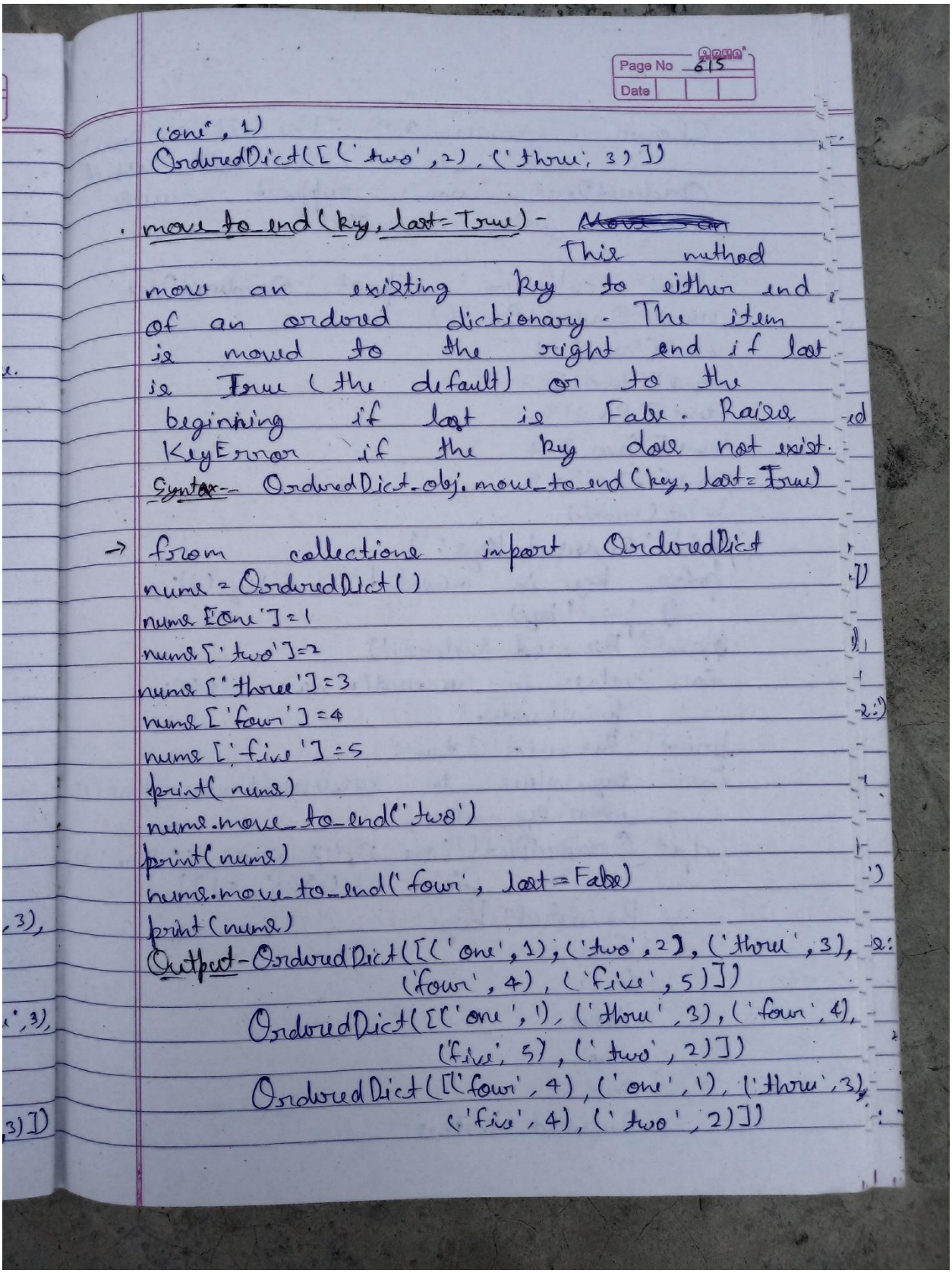
New in version 3.1 -

- popitem(last = True) - The popitem() method for ordered dictionaries returns and removes a (key, value) pair. The pairs are return in LIFO order if last is True or FIFO order if false.

Syntax - Ordered_dict = obj.popitem(last = True).

→
```
from collections import OrderedDict
nums = OrderDict()
nums['one'] = 1
nums['two'] = 2
nums['three'] = 3
nums['four'] = 4
nums['five'] = 5
print(nums)
print(nums.popitem())                    (remove).
print(nums)                              (random).
print(nums.popitem(last = True))         (last)
print(nums)
print(nums.popitem(last = False))        (first)
print(nums)
```

Output - OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)])
('five', 5)
OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4)])
('four', 4)
OrderedDict([('one', 1), ('two', 2), ('three', 3)])

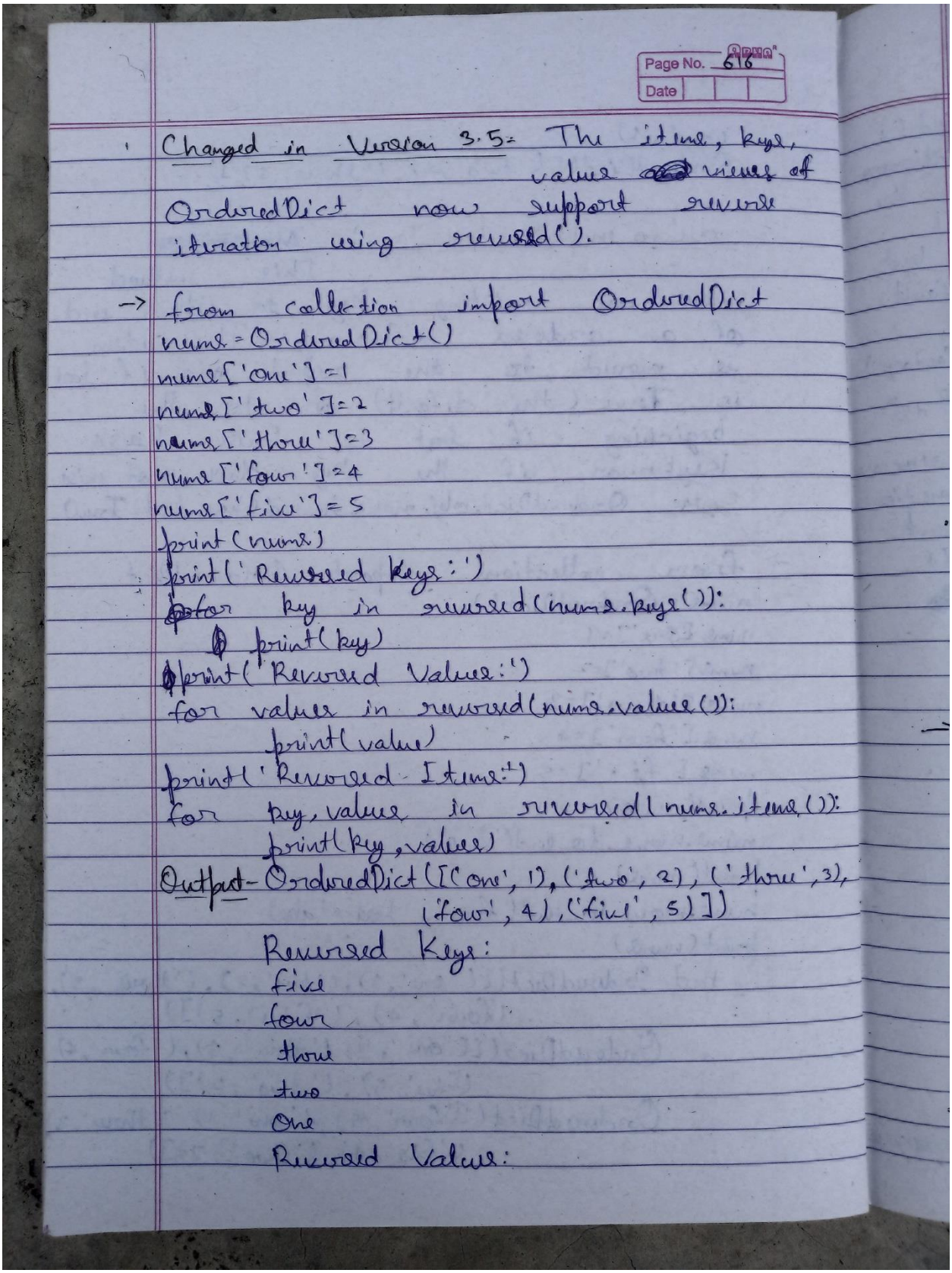


Page No 615
Date

('one', 1)
OrderedDict([('two', 2), ('three', 3)])

. move_to_end(key, last=True) - This method move an existing key to either end of an ordered dictionary. The item is moved to the right end if last is True (the default) or to the beginning if last is False. Raise KeyError if the key does not exist.
Syntax - OrderedDict_obj.move_to_end(key, last=True)

→ from collections import OrderedDict
nums = OrderedDict()
nums['one'] = 1
nums['two'] = 2
nums['three'] = 3
nums['four'] = 4
nums['five'] = 5
print(nums)
nums.move_to_end('two')
print(nums)
nums.move_to_end('four', last=False)
print(nums)
Output - OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)])
OrderedDict([('one', 1), ('three', 3), ('four', 4), ('five', 5), ('two', 2)])
OrderedDict([('four', 4), ('one', 1), ('three', 3), ('five', 4), ('two', 2)])



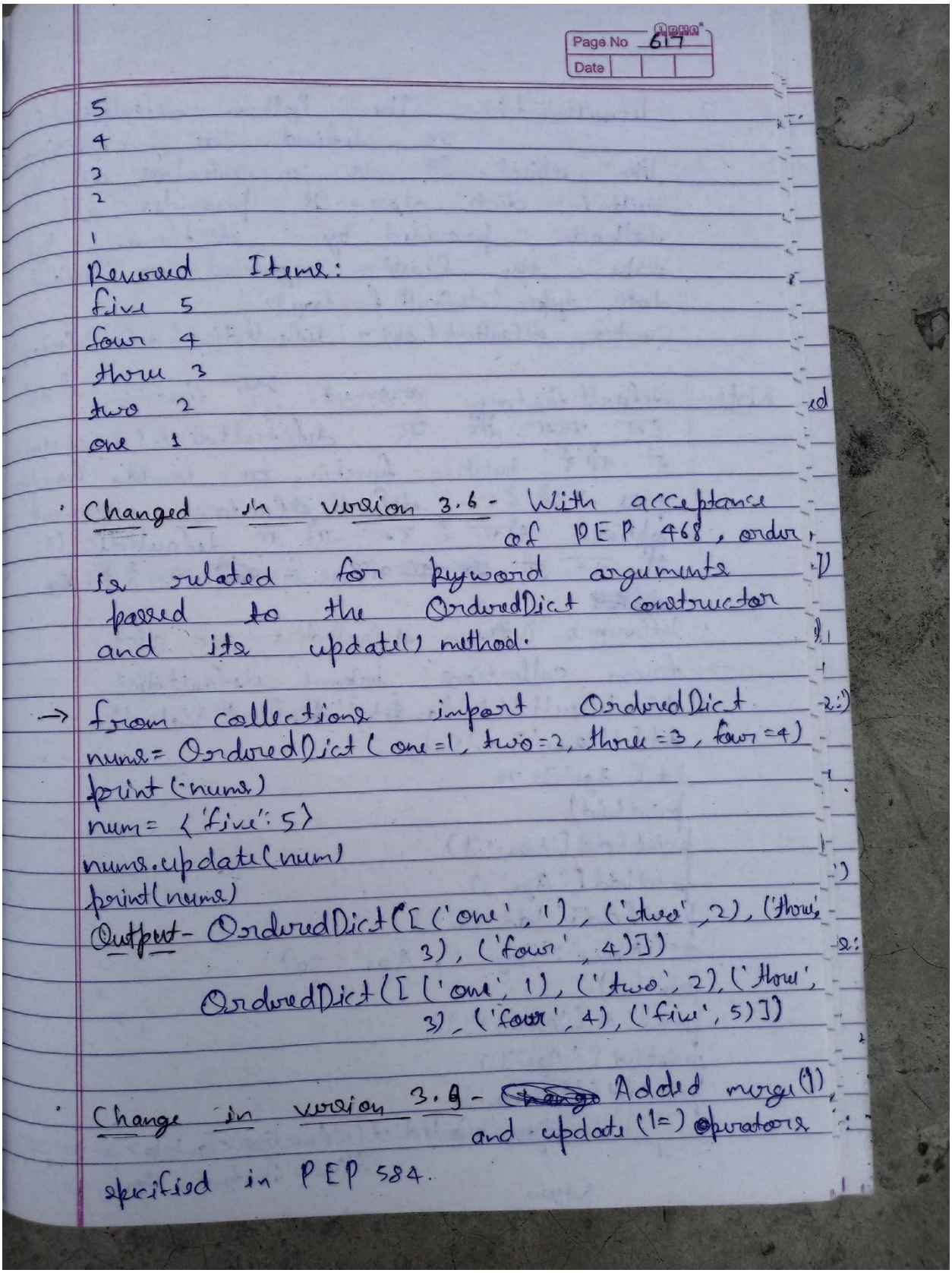
Page No. 616
Date

Changed in Version 3.5: The items, keys, values views of OrderedDict now support reverse iteration using reversed().

→
```
from collection import OrderedDict
nums = OrderedDict()
nums['one'] = 1
nums['two'] = 2
nums['three'] = 3
nums['four'] = 4
nums['five'] = 5
print(nums)
print('Reversed Keys:')
for key in reversed(nums.keys()):
    print(key)
print('Reversed Values:')
for values in reversed(nums.values()):
    print(value)
print('Reversed Items:')
for key, value in reversed(nums.items()):
    print(key, values)
```

Output- OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)])

Reversed Keys:
five
four
three
two
One
Reversed Values:



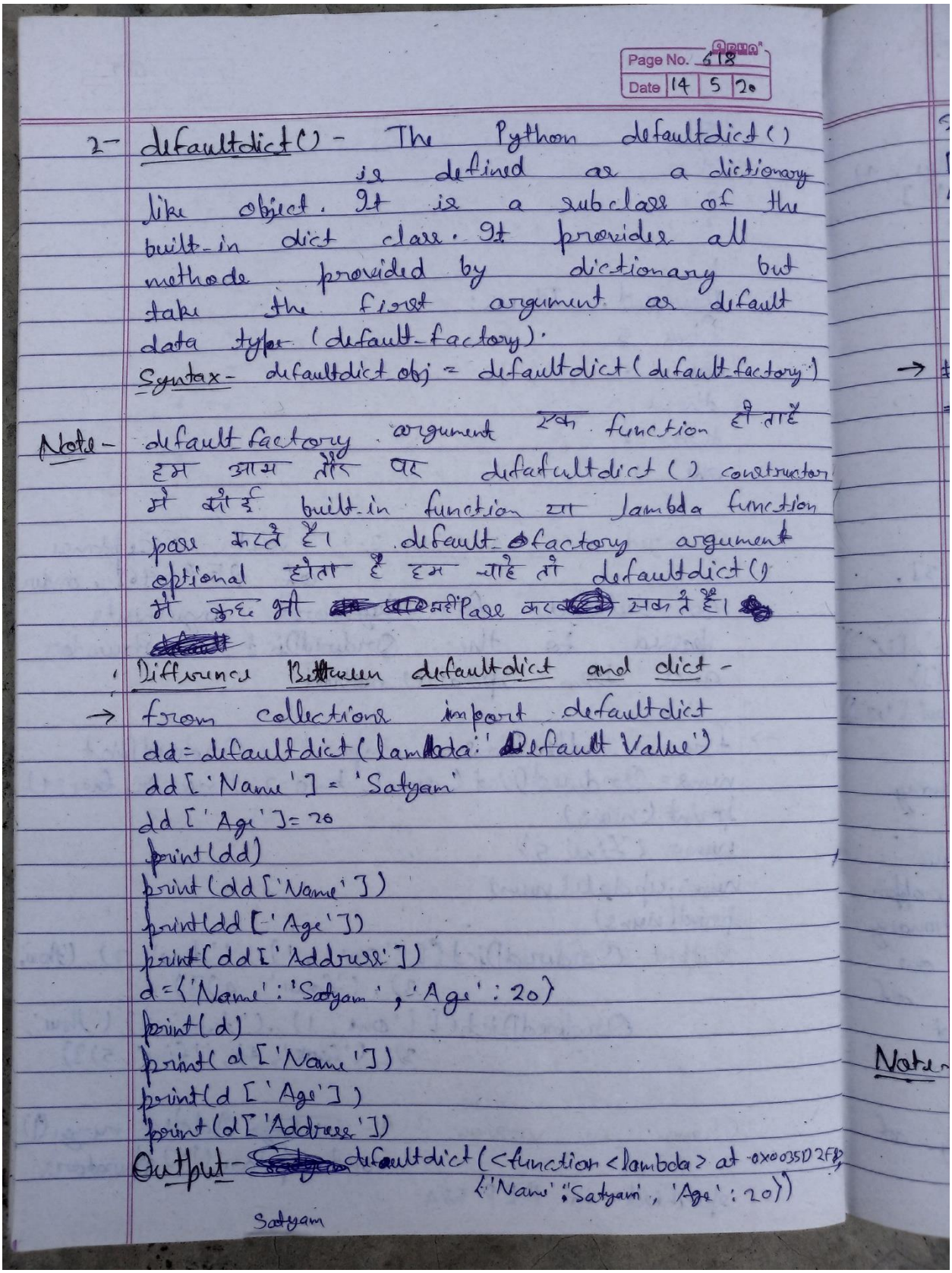
Page No 617
Date

5
4
3
2
1
Reversed Items:
five 5
four 4
three 3
two 2
one 1

• Changed in version 3.6 - With acceptance of PEP 468, order is related for keyword arguments passed to the OrderedDict constructor and its update() method.

→ from collections import OrderedDict
nums = OrderedDict(one=1, two=2, three=3, four=4)
print(nums)
num = {'five': 5}
nums.update(num)
print(nums)
Output - OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4)])
OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)])

• Change in version 3.9 - ~~Change~~ Added merge (|) and update (|=) operators specified in PEP 584.



Page No. 418
Date 14 5 20

2- defaultdict() - The Python defaultdict() is defined as a dictionary like object. It is a subclass of the built-in dict class. It provides all methods provided by dictionary but take the first argument as default data type (default_factory).

Syntax- defaultdict obj = defaultdict(default_factory)

Note- default_factory argument एक function होता है हम आम तौर पर defaultdict() constructor में तो built-in function या lambda function pass करते है। default_factory argument optional होता है हम चाहे तो defaultdict() में कुछ भी नहीं Pass कर सकते है।

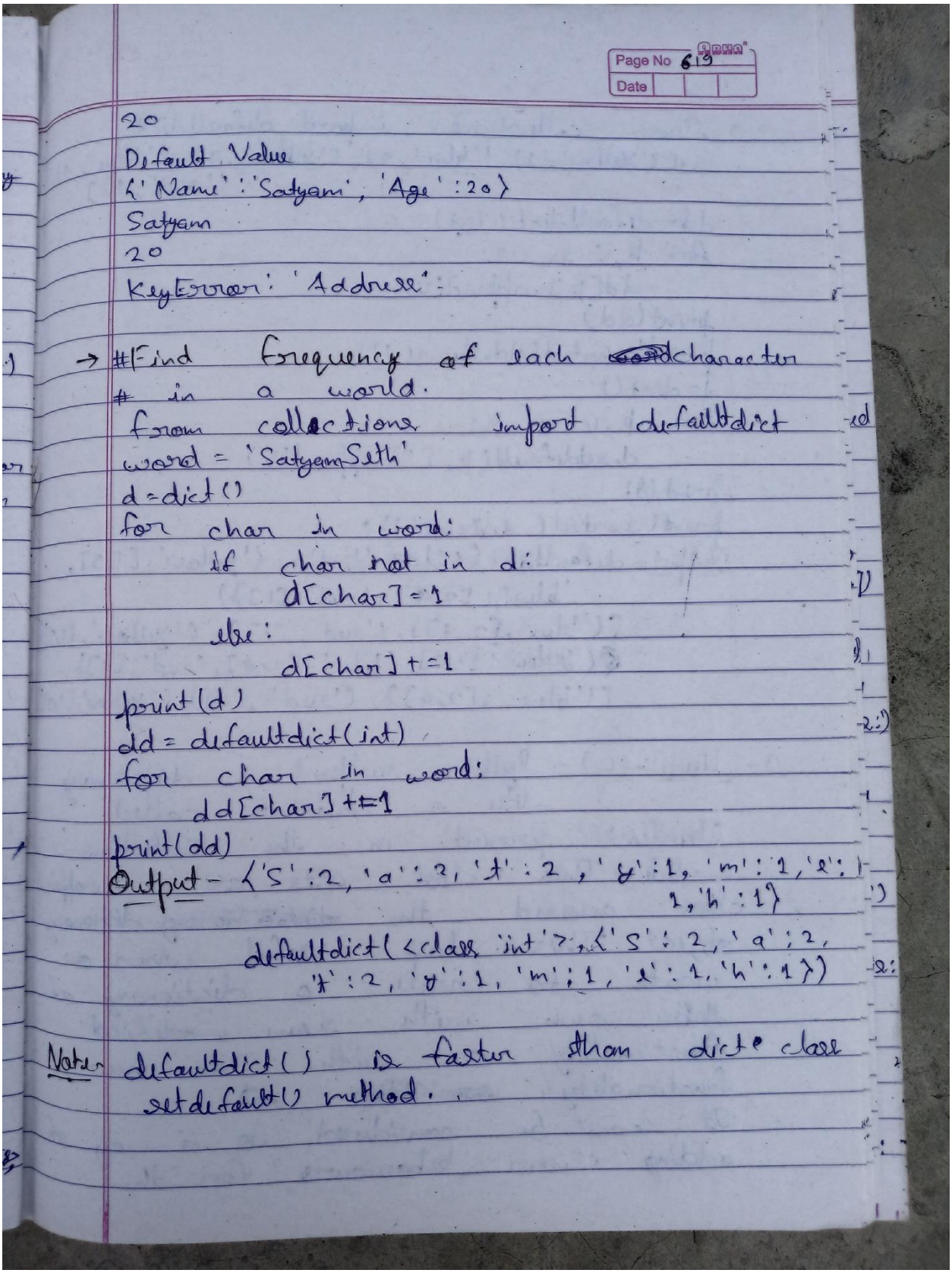
Difference Between defaultdict and dict -

→
```
from collections import defaultdict
dd = defaultdict(lambda: 'Default Value')
dd['Name'] = 'Satyam'
dd['Age'] = 20
print(dd)
print(dd['Name'])
print(dd['Age'])
print(dd['Address'])
d = {'Name': 'Satyam', 'Age': 20}
print(d)
print(d['Name'])
print(d['Age'])
print(d['Address'])
```

Output- defaultdict(<function <lambda> at 0x0035D2F8>, {'Name': 'Satyam', 'Age': 20})

Satyam

Note-



Page No 619
Date

20
Default Value
{'Name': 'Satyam', 'Age': 20}
Satyam
20
KeyError: 'Address'

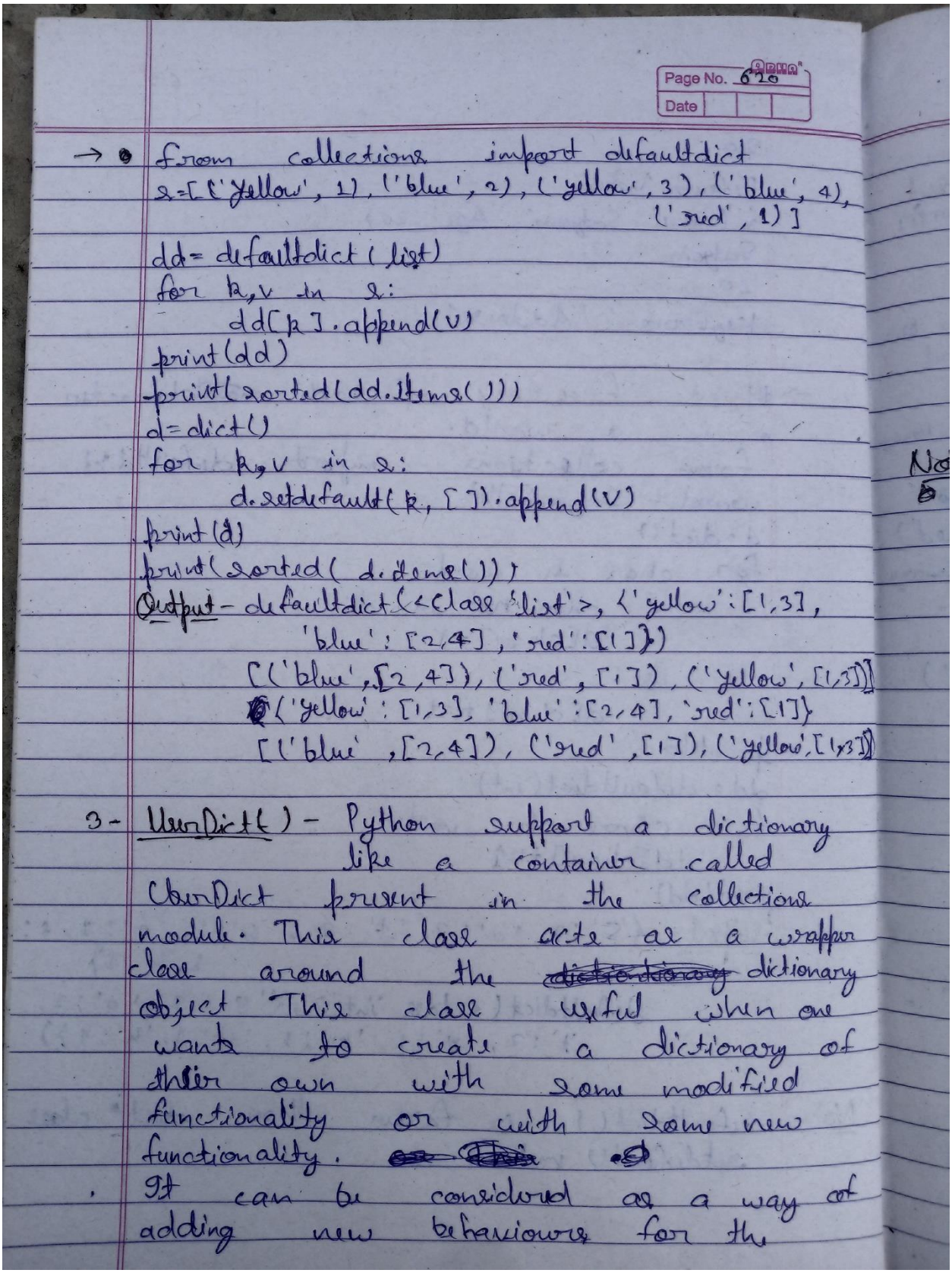
→ #Find frequency of each character in a world.

```
from collections import defaultdict
word = 'SatyamSeth'
d = dict()
for char in word:
    if char not in d:
        d[char] = 1
    else:
        d[char] += 1
print(d)
dd = defaultdict(int)
for char in word:
    dd[char] += 1
print(dd)
```

Output - {'S': 2, 'a': 2, 't': 2, 'y': 1, 'm': 1, 'e': 1, 'h': 1}

defaultdict(<class 'int'>, {'S': 2, 'a': 2, 't': 2, 'y': 1, 'm': 1, 'e': 1, 'h': 1})

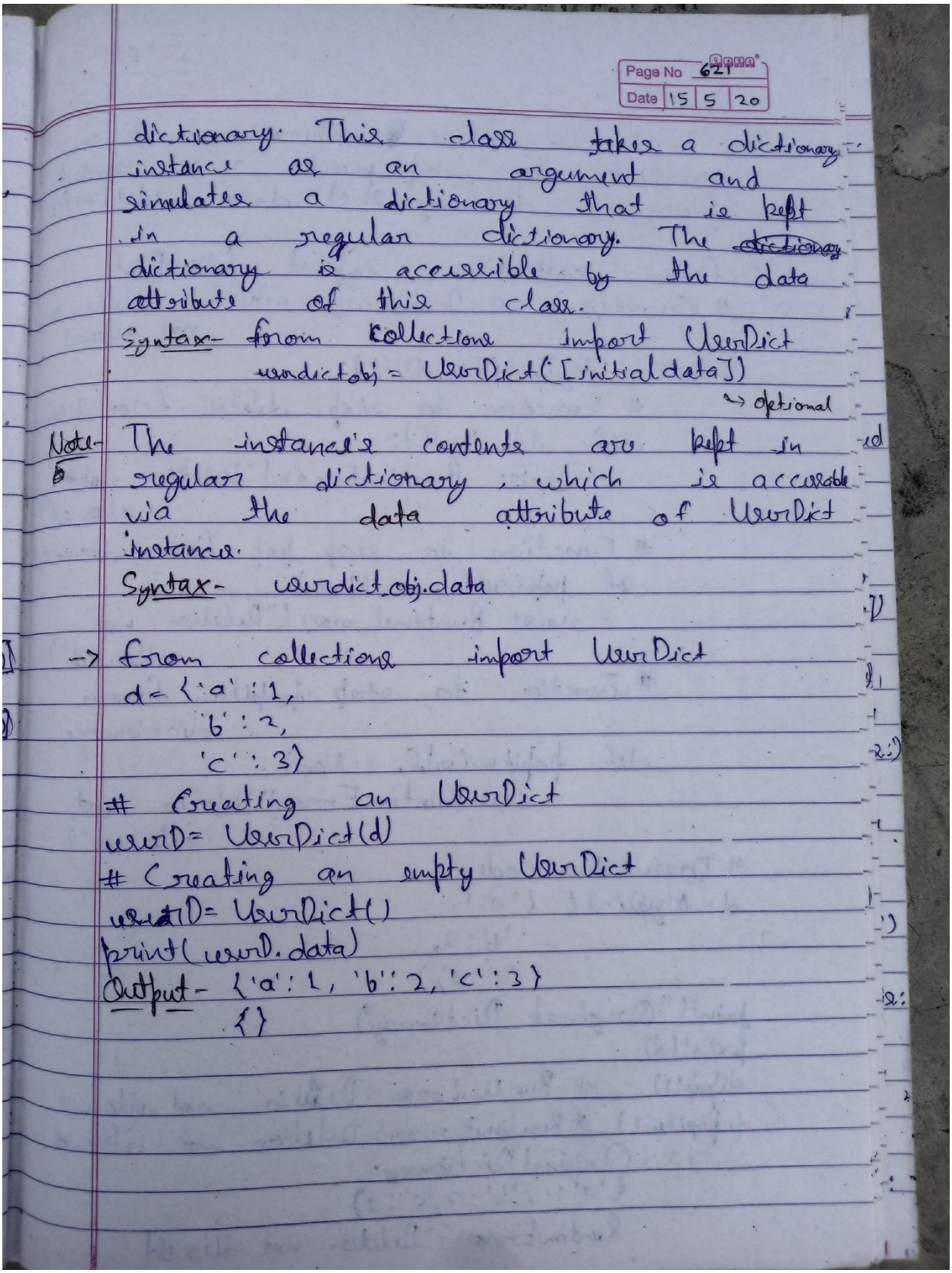
Note: defaultdict() is faster than dict class setdefault() method.



→ 2
```
from collections import defaultdict
s=[('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4),
                                ('red', 1)]
dd= defaultdict(list)
for k,v in s:
    dd[k].append(v)
print(dd)
print(sorted(dd.items()))
d=dict()
for k,v in s:
    d.setdefault(k, []).append(v)
print(d)
print(sorted(d.items()))
```
Output - defaultdict(<class 'list'>, {'yellow': [1,3], 'blue': [2,4], 'red': [1]})
[('blue',[2,4]), ('red', [1]), ('yellow', [1,3])]
{'yellow': [1,3], 'blue': [2,4], 'red': [1]}
[('blue' ,[2,4]), ('red' ,[1]), ('yellow',[1,3])]

3- UserDict() - Python support a dictionary like a container called UserDict present in the collections module. This class acts as a wrapper class around the ~~dictionary~~ dictionary object This class useful when one wants to create a dictionary of their own with some modified functionality or with some new functionality.

It can be considered as a way of adding new behaviours for the



Page No 621
Date 15 5 20

dictionary. This class takes a dictionary instance as an argument and simulates a dictionary that is kept in a regular dictionary. The ~~dictionary~~ dictionary is accessible by the data attribute of this class.

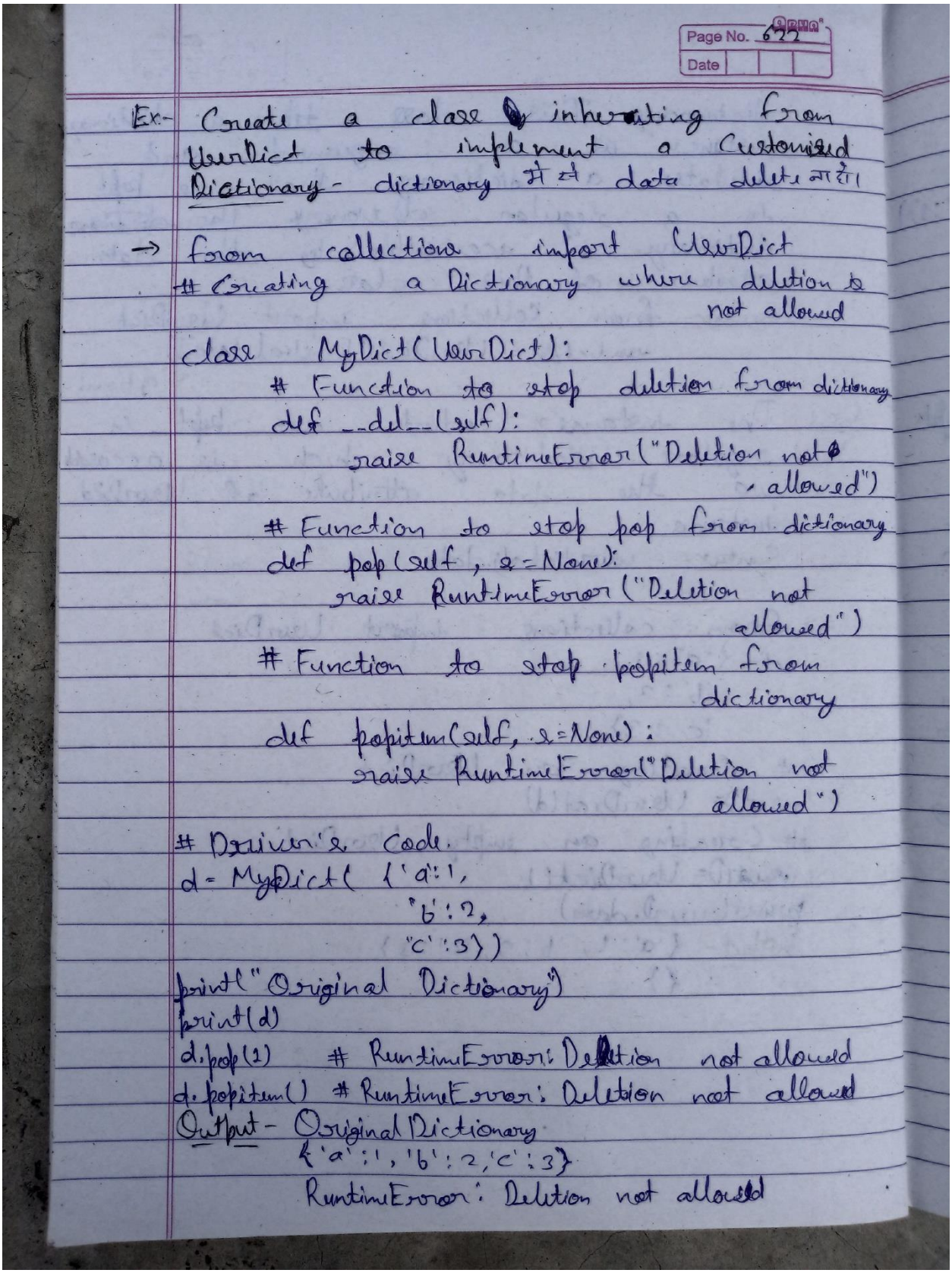
Syntax- from Collections import UserDict
userdictobj = UserDict([initialdata])
↳ optional

Note- The instance's contents are kept in regular dictionary, which is accessible via the data attribute of UserDict instance.

Syntax- userdict_obj.data

-> from collections import UserDict
d = {'a': 1,
'b': 2,
'c': 3}
# Creating an UserDict
userD = UserDict(d)
# Creating an empty UserDict
userD = UserDict()
print(userD.data)

Output- {'a': 1, 'b': 2, 'c': 3}
{}



Page No. 622
Date

Ex- Create a class by inheriting from UserDict to implement a Customised Dictionary - dictionary में से data delete न हो।

→ from collections import UserDict
Creating a Dictionary where deletion is not allowed

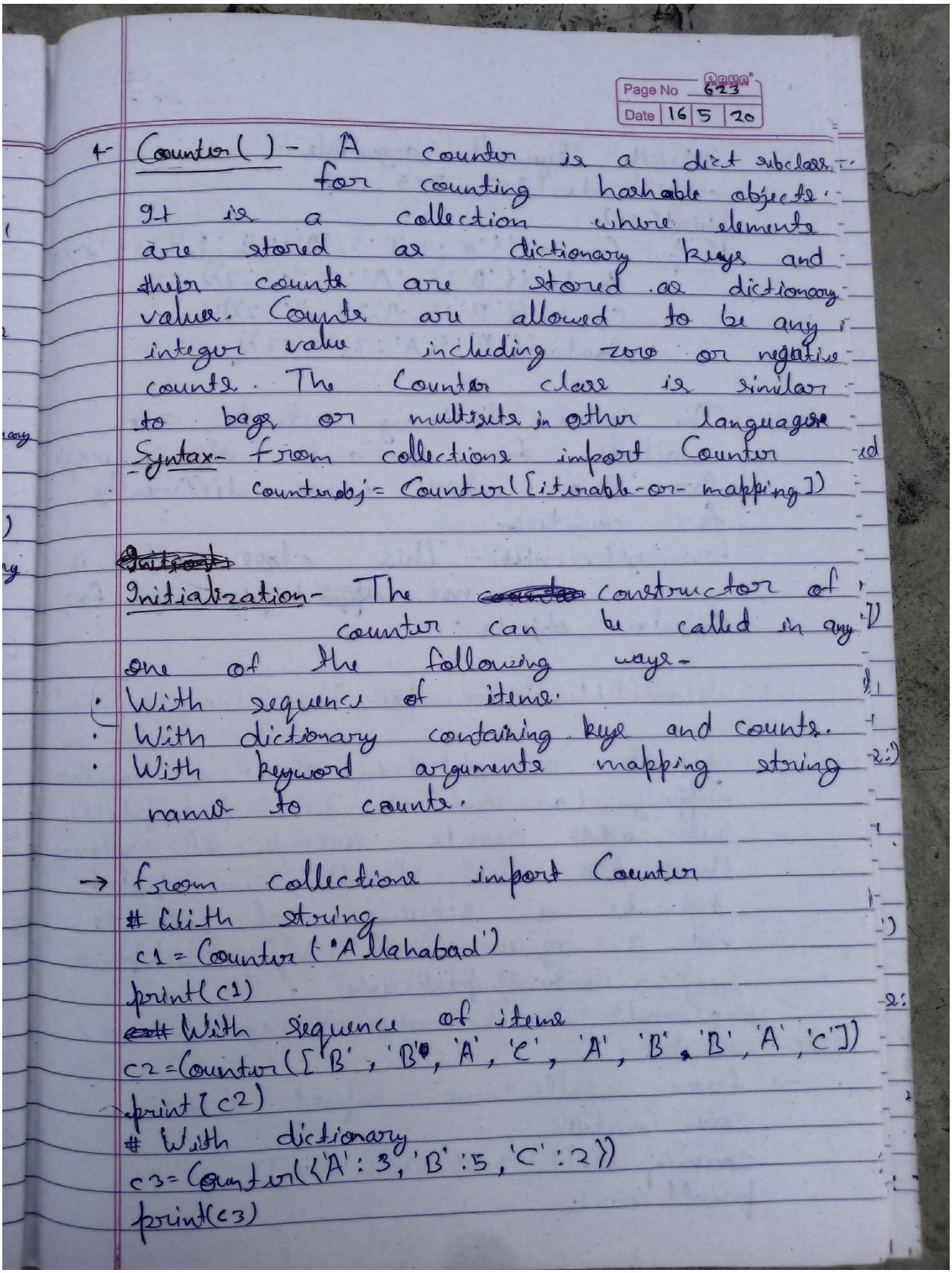
class MyDict(UserDict):
    # Function to stop deletion from dictionary
    def __del__(self):
        raise RuntimeError("Deletion not allowed")

    # Function to stop pop from dictionary
    def pop(self, s=None):
        raise RuntimeError("Deletion not allowed")

    # Function to stop popitem from dictionary
    def popitem(self, s=None):
        raise RuntimeError("Deletion not allowed")

# Driver's code
d = MyDict({'a':1,
            'b':2,
            'c':3})
print("Original Dictionary")
print(d)
d.pop(1) # RuntimeError: Deletion not allowed
d.popitem() # RuntimeError: Deletion not allowed

Output - Original Dictionary
{'a':1, 'b':2, 'c':3}
RuntimeError: Deletion not allowed



Page No 623
Date 16 5 20

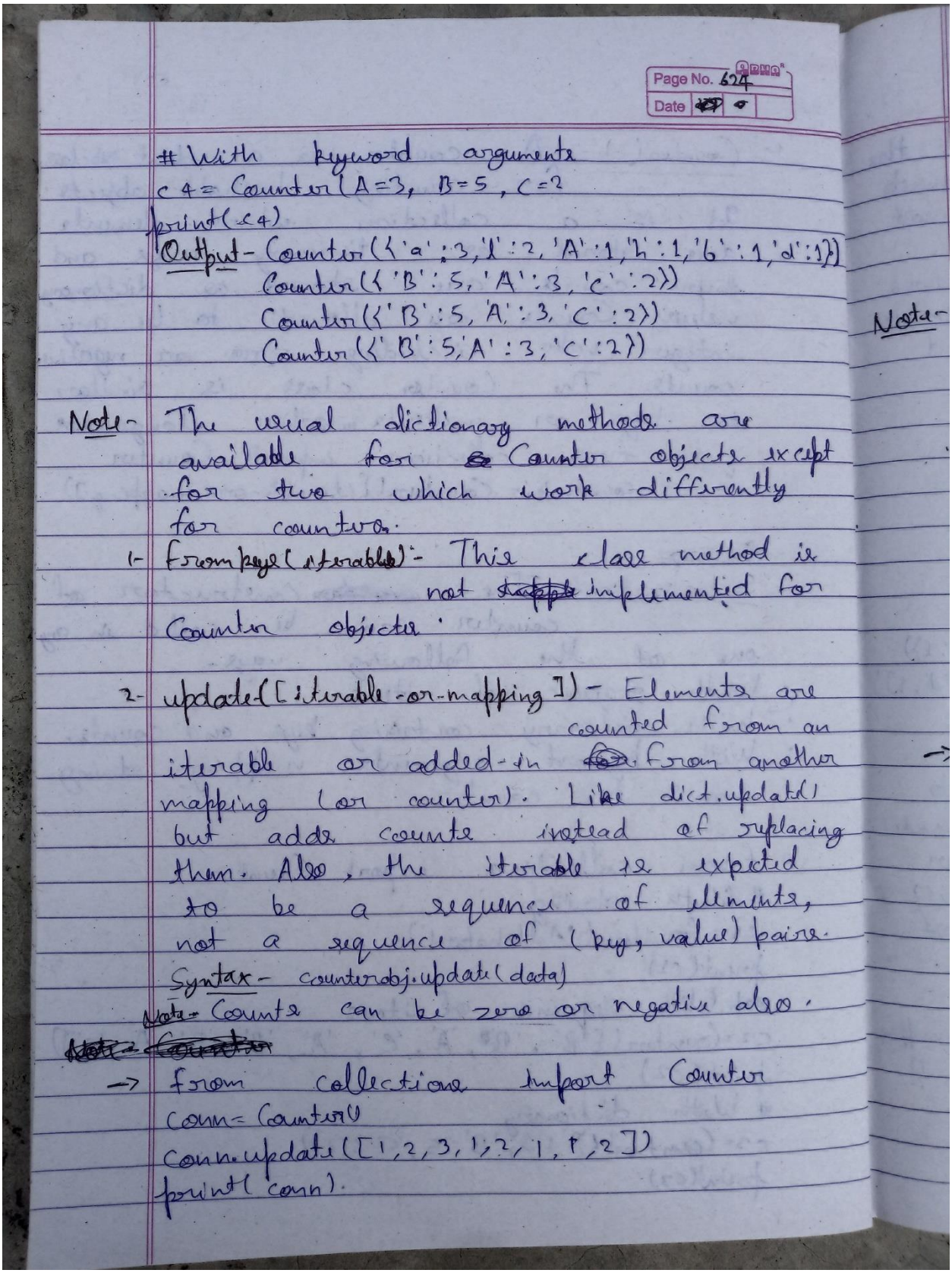
+ Counter() - A counter is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary value. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Syntax- from collections import Counter
countobj = Counter([iterable-or-mapping])

Initialization- The ~~counter~~ constructor of counter can be called in any one of the following ways-

- With sequence of items.
- With dictionary containing keys and counts.
- With keyword arguments mapping string names to counts.

→ from collections import Counter
# With string
c1 = Counter('Allahabad')
print(c1)
# With sequence of items
c2 = Counter(['B', 'B', 'A', 'C', 'A', 'B', 'B', 'A', 'C'])
print(c2)
# With dictionary
c3 = Counter({'A': 3, 'B': 5, 'C': 2})
print(c3)



Page No. 624
Date

# With keyword arguments
c4 = Counter(A=3, B=5, C=2)
print(c4)
Output - Counter({'a': 3, 'l': 2, 'A': 1, 'h': 1, 'b': 1, 'd': 1})
Counter({'B': 5, 'A': 3, 'C': 2})
Counter({'B': 5, 'A': 3, 'C': 2})
Counter({'B': 5, 'A': 3, 'C': 2})

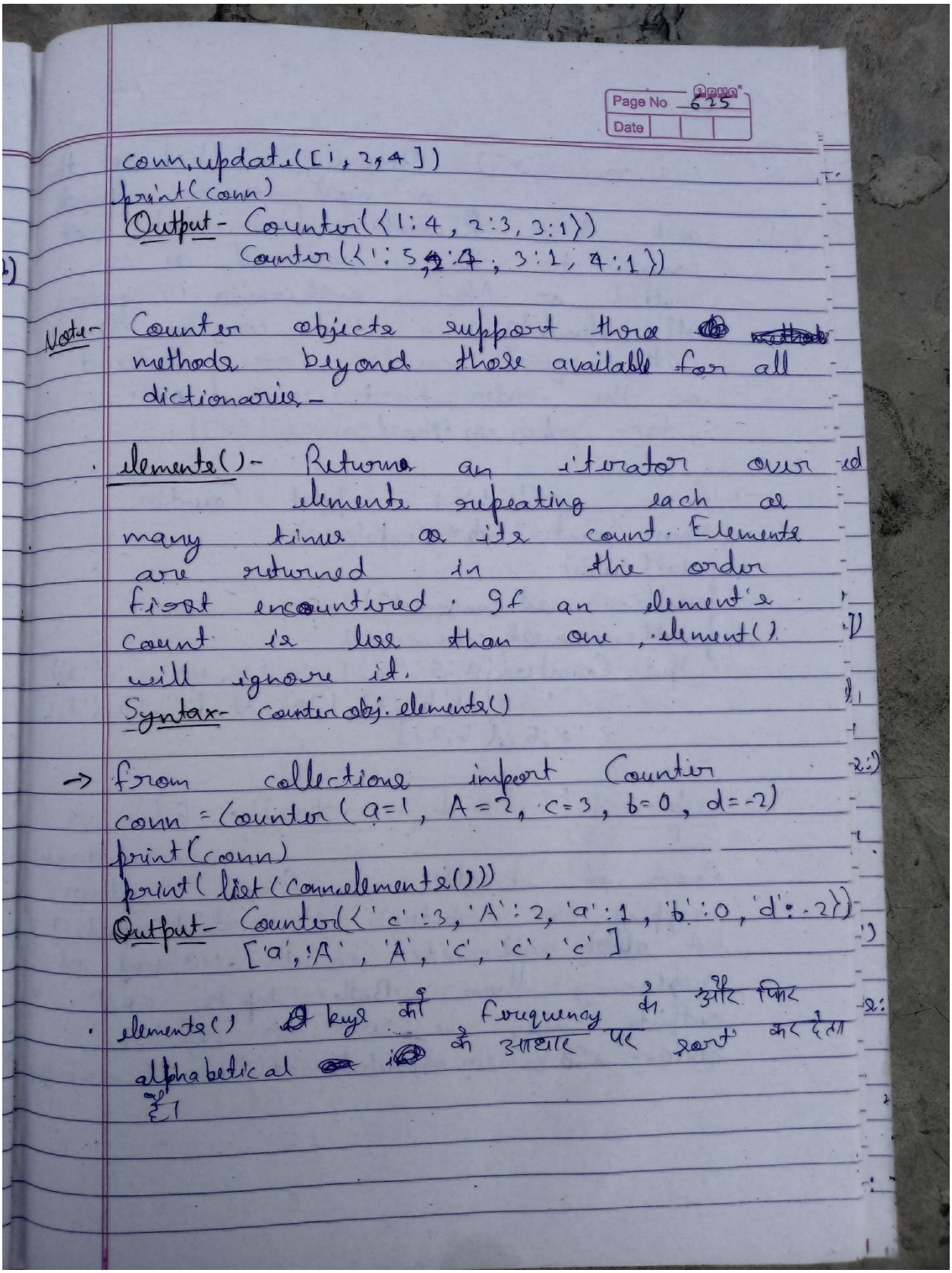
Note - The usual dictionary methods are available for Counter objects except for two which work differently for counters.

1- fromkeys(iterable) - This class method is not implemented for Counter objects.

2- update([iterable-or-mapping]) - Elements are counted from an iterable or added-in from another mapping (or counter). Like dict.update() but adds counts instead of replacing them. Also, the iterable is expected to be a sequence of elements, not a sequence of (key, value) pairs.
Syntax - counterobj.update(data)
Note - Counts can be zero or negative also.

-> from collections import Counter
coun = Counter()
coun.update([1, 2, 3, 1, 2, 1, 1, 2])
print(coun)

Note -



Page No 625
Date

```
conn.update([1, 2, 4])
print(conn)
```

Output- Counter({1:4, 2:3, 3:1})
Counter({1:5, 2:4, 3:1, 4:1})

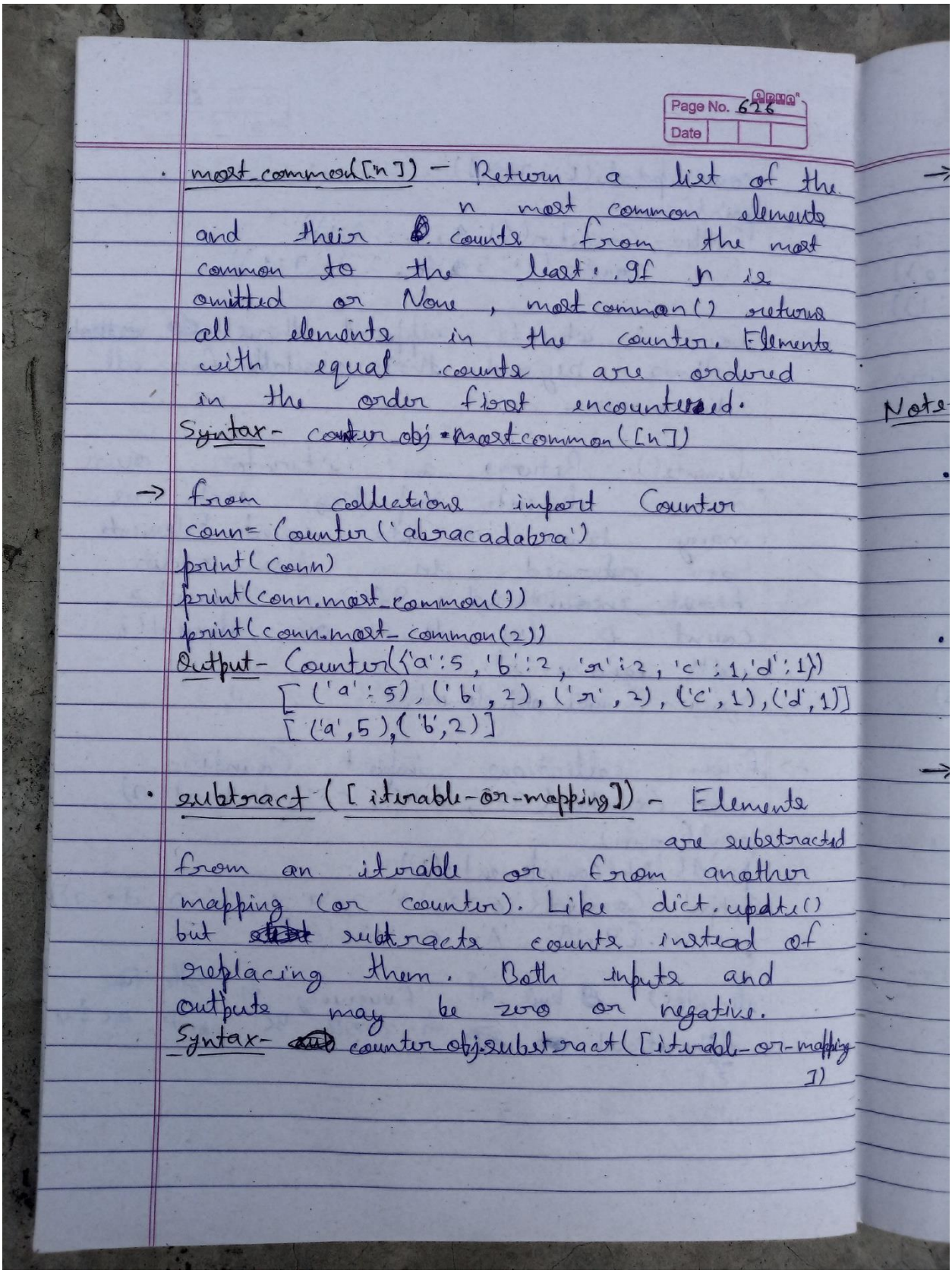
Note- Counter objects support three methods beyond those available for all dictionaries -

- **elements()**- Returns an iterator over elements repeating each as many times as its count. Elements are returned in the order first encountered. If an element's count is less than one, .element() will ignore it.

Syntax- Counterobj.elements()

→
```
from collections import Counter
conn = Counter(a=1, A=2, c=3, b=0, d=-2)
print(conn)
print(list(conn.elements()))
```
Output- Counter({'c':3, 'A':2, 'a':1, 'b':0, 'd':-2})
['a', 'A', 'A', 'c', 'c', 'c']

- elements() key को frequency के आधार पर फिर alphabetical के आधार पर sort कर देता है।



Page No. 626
Date

- most_common([n]) – Return a list of the n most common elements and their counts from the most common to the least. If n is omitted or None, most_common() returns all elements in the counter. Elements with equal counts are ordered in the order first encountered.

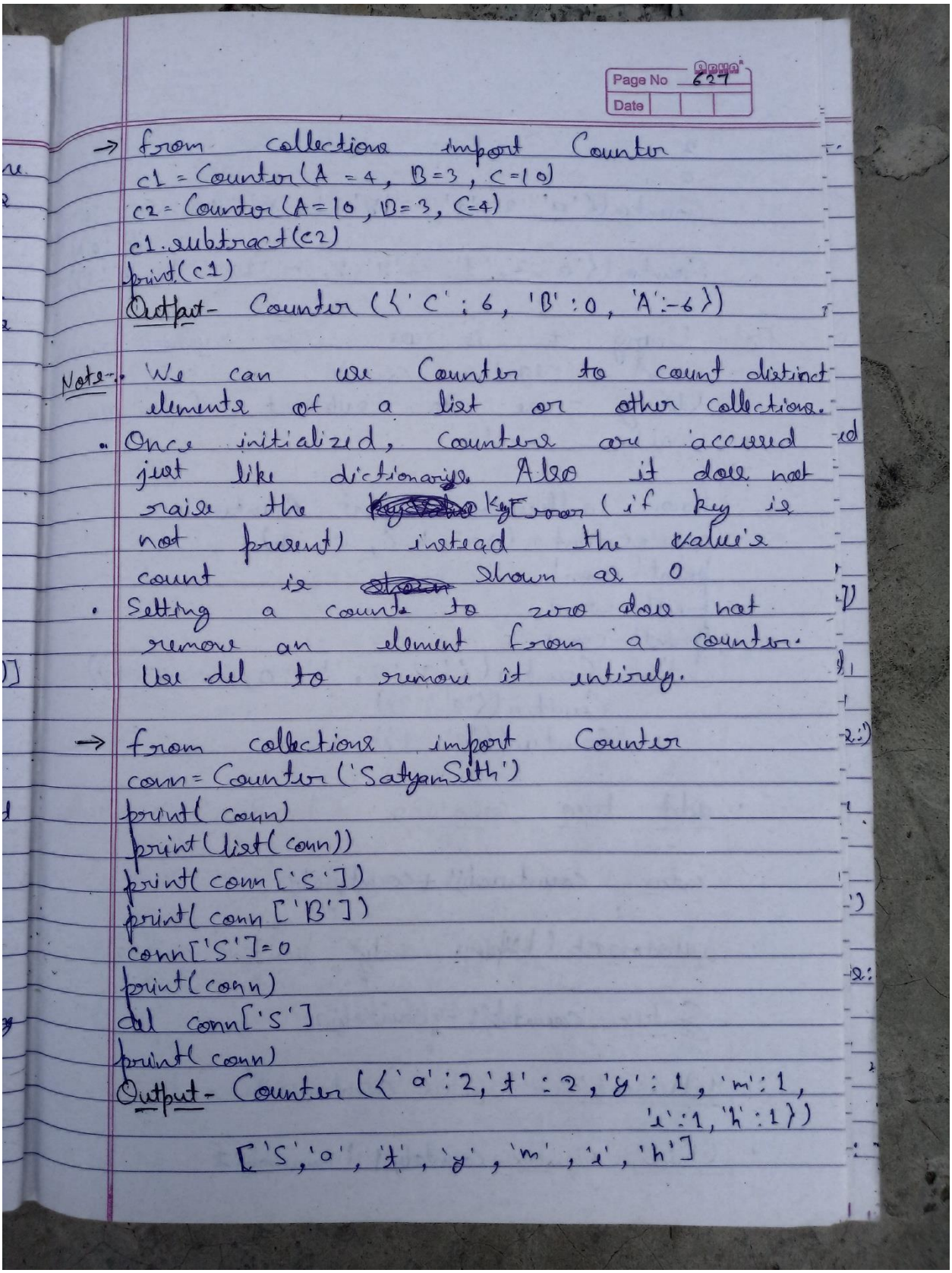
Syntax- counter_obj.most_common([n])

Note

→ from collections import Counter
conn = Counter('abracadabra')
print(conn)
print(conn.most_common())
print(conn.most_common(2))
Output- Counter({'a':5, 'b':2, 'r':2, 'c':1, 'd':1})
[('a':5), ('b',2), ('r',2), ('c',1), ('d',1)]
[('a',5), ('b',2)]

- subtract([iterable-or-mapping]) – Elements are subtracted from an iterable or from another mapping (or counter). Like dict.update() but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

Syntax- counter_obj.subtract([iterable-or-mapping])

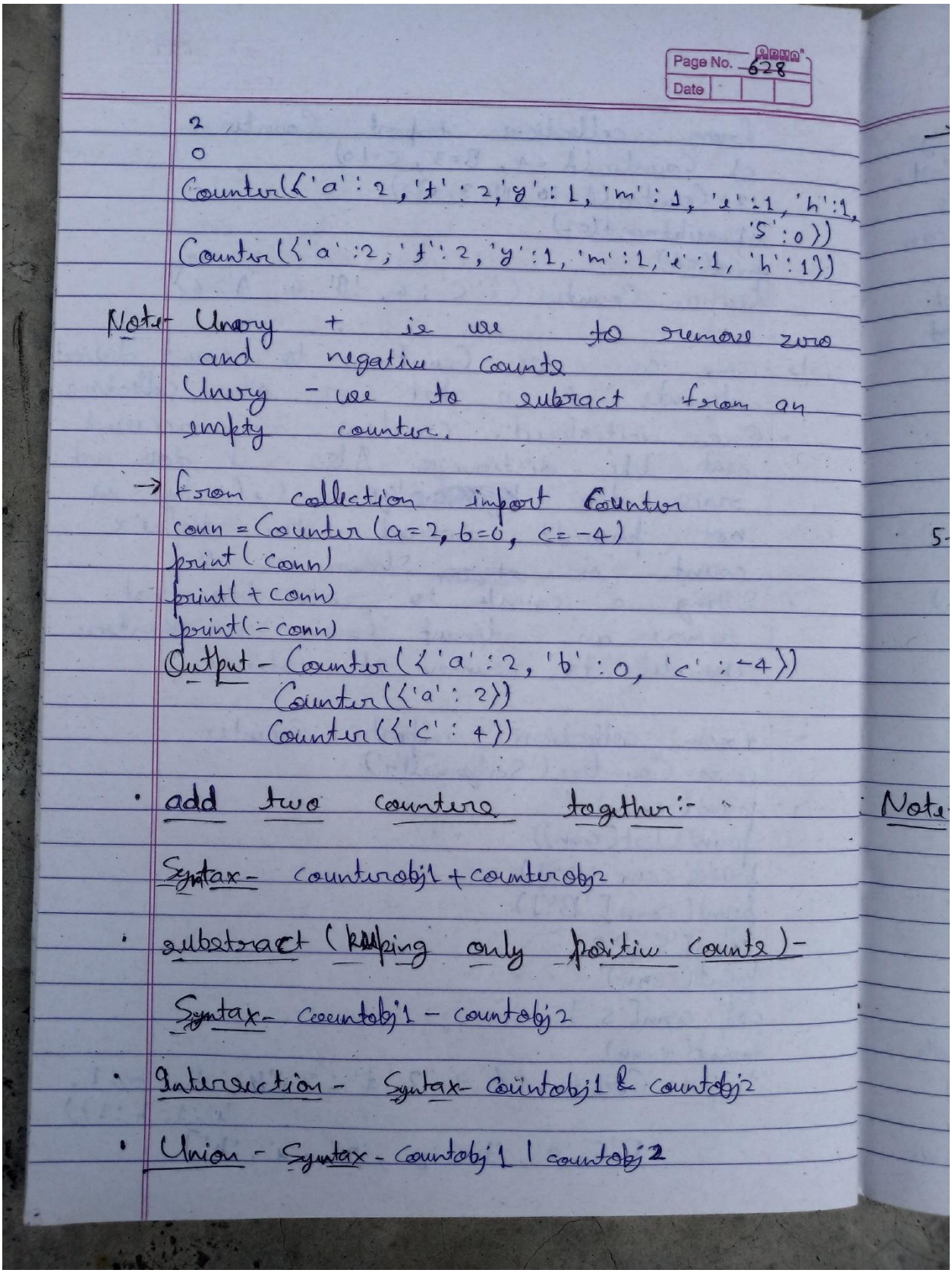


Page No 627
Date

→ from collections import Counter
c1 = Counter(A = 4, B = 3, C = 10)
c2 = Counter(A = 10, B = 3, C = 4)
c1.subtract(c2)
print(c1)
Output- Counter({'C': 6, 'B': 0, 'A': -6})

Note-
- We can use Counter to count distinct elements of a list or other collections.
- Once initialized, counters are accessed just like dictionaries. Also it does not raise the KeyError (if key is not present) instead the value's count is shown as 0
- Setting a count to zero does not remove an element from a counter. Use del to remove it entirely.

→ from collections import Counter
conn = Counter('SatyamSeth')
print(conn)
print(list(conn))
print(conn['s'])
print(conn['B'])
conn['S'] = 0
print(conn)
del conn['s']
print(conn)
Output- Counter({'a': 2, 't': 2, 'y': 1, 'm': 1, 'e': 1, 'h': 1})
['S', 'a', 't', 'y', 'm', 'e', 'h']



Page No. 628
Date

2
0
Counter({'a': 2, 't': 2, 'y': 1, 'm': 1, 'e': 1, 'h': 1, '5': 0})
Counter({'a': 2, 't': 2, 'y': 1, 'm': 1, 'e': 1, 'h': 1})

Note- Unary + is use to remove zero and negative counts
Unary - use to subtract from an empty counter.

→ from collection import Counter
conn = Counter(a=2, b=0, c=-4)
print(conn)
print(+conn)
print(-conn)
Output - Counter({'a': 2, 'b': 0, 'c': -4})
Counter({'a': 2})
Counter({'c': 4})

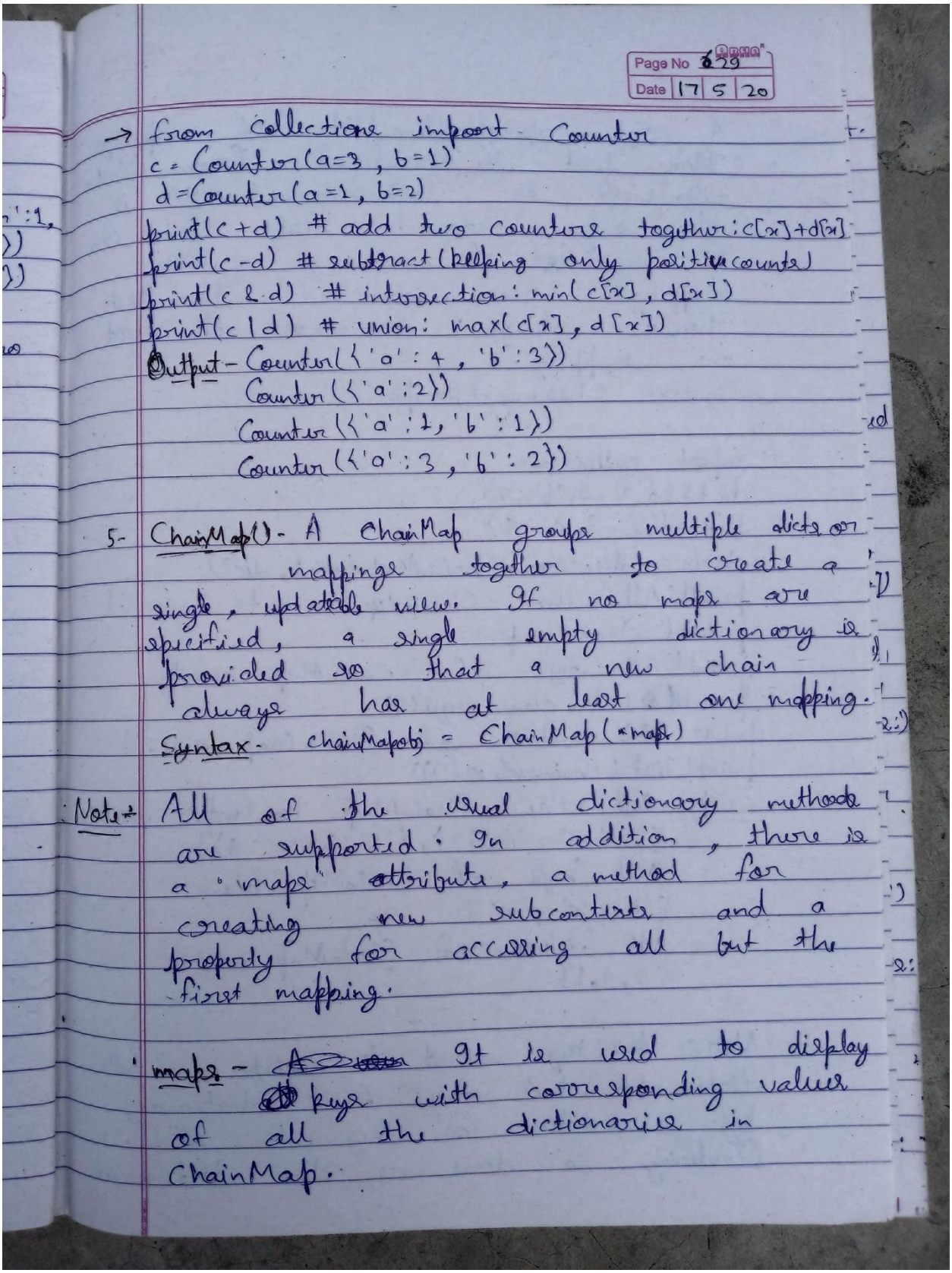
- add two counters togther:-

Syntax- counterobj1 + counterobj2

- substract (keeping only positive counts)-

Syntax- Coountobj1 - countobj2

- Intersection - Syntax- Countobj1 & countobj2
- Union - Syntax - Countobj1 | countobj2



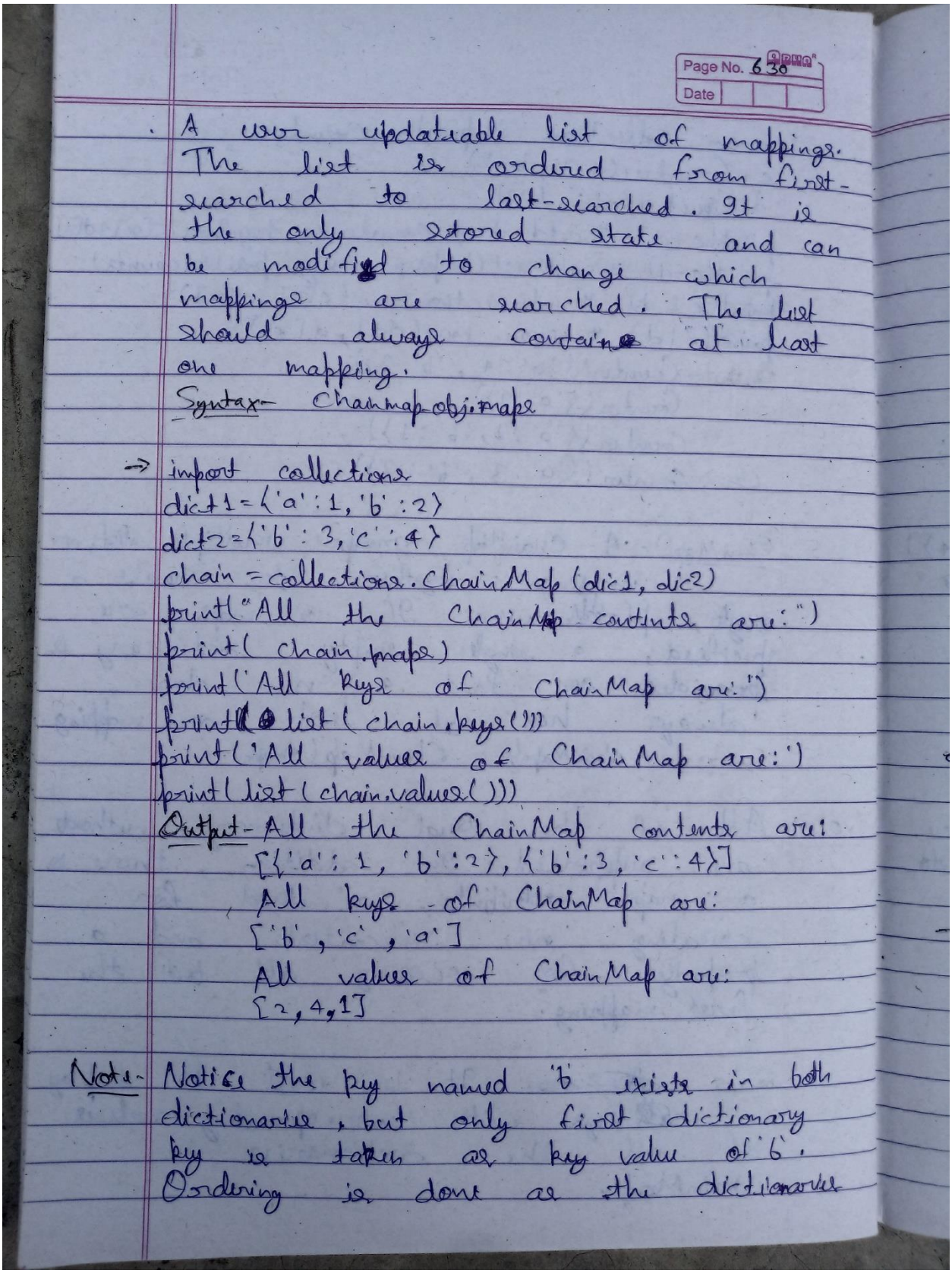
Page No 29
Date 17 5 20

→ from collections import Counter
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
print(c+d) # add two counters together: c[x]+d[x]
print(c-d) # subtract (keeping only positive counts)
print(c & d) # intersection: min(c[x], d[x])
print(c | d) # union: max(c[x], d[x])
Output - Counter({'a': 4, 'b': 3})
Counter({'a': 2})
Counter({'a': 1, 'b': 1})
Counter({'a': 3, 'b': 2})

5- ChainMap() - A ChainMap groups multiple dicts or mappings together to create a single, updatable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.
Syntax - chainMapobj = ChainMap(*maps)

Note:- All of the usual dictionary methods are supported. In addition, there is a 'maps' attribute, a method for creating new subcontexts and a property for accessing all but the first mapping.

maps - It is used to display keys with corresponding values of all the dictionaries in ChainMap.



Page No. 6 30
Date

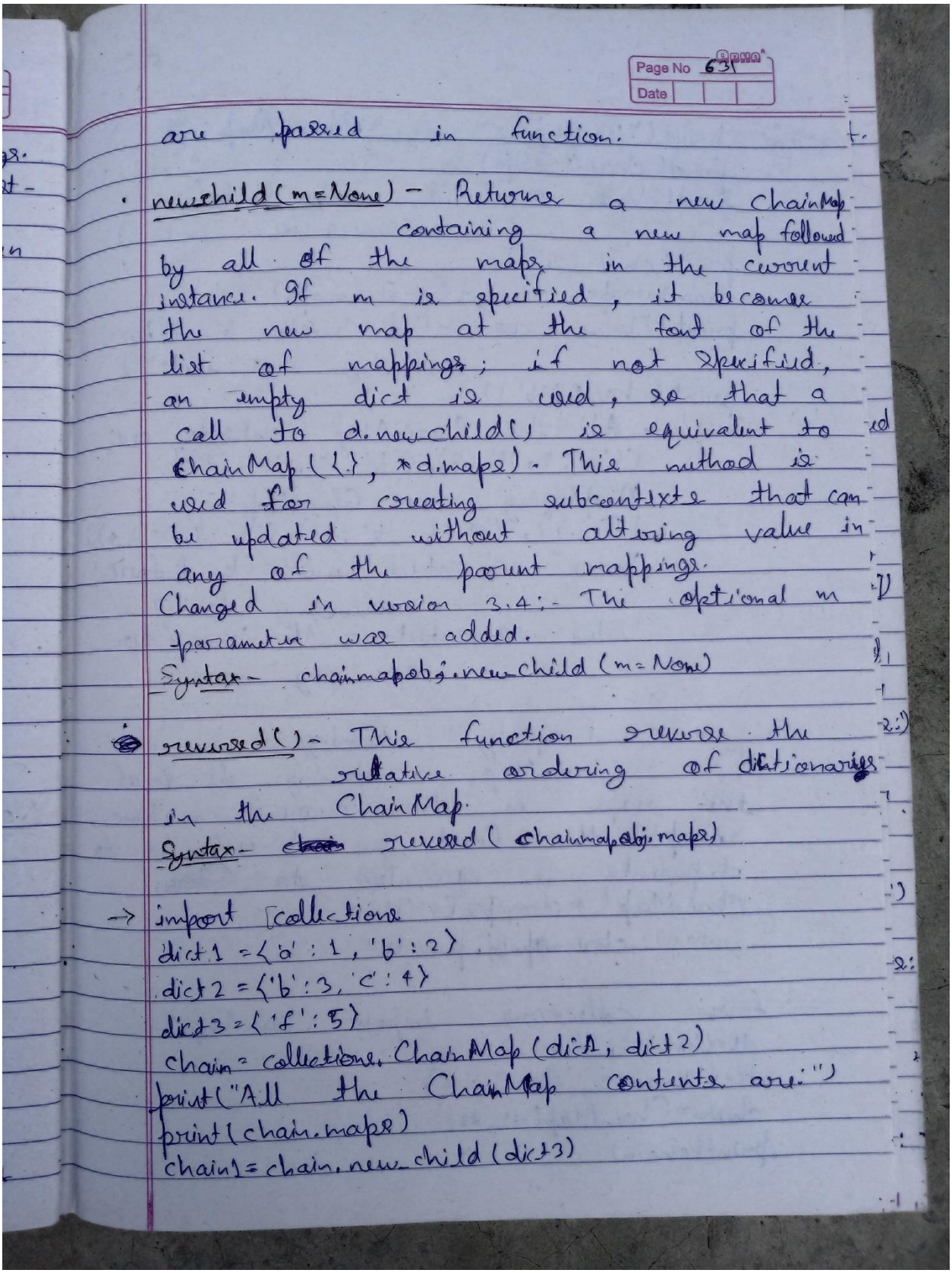
. A user updatable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

Syntax- Chainmap.obj.maps

→
```
import collections
dict1={'a':1,'b':2}
dict2={'b':3,'c':4}
chain=collections.ChainMap(dic1,dic2)
print("All the ChainMap contents are:")
print(chain.maps)
print('All keys of ChainMap are:")
print(list(chain.keys()))
print('All values of ChainMap are:')
print(list(chain.values()))
```

Output- All the ChainMap contents are:
[{'a':1,'b':2},{'b':3,'c':4}]
All keys of ChainMap are:
['b','c','a']
All values of ChainMap are:
[2,4,1]

Note- Notice the key named 'b' exists in both dictionaries, but only first dictionary key is taken as key value of 'b'. Ordering is done as the dictionaries

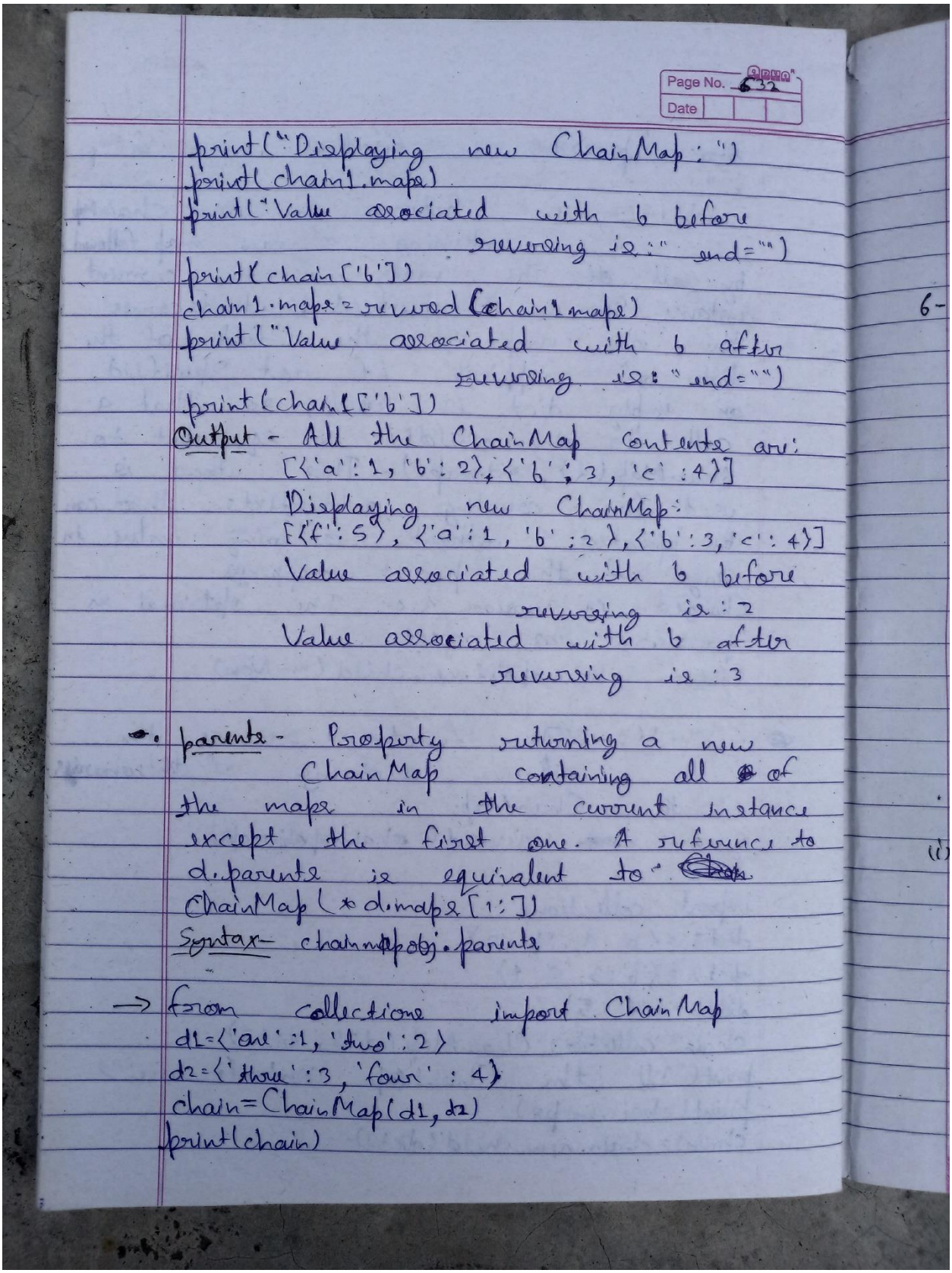


are passed in function.

- newchild(m=None) - Returns a new ChainMap containing a new map followed by all of the maps in the current instance. If m is specified, it becomes the new map at the font of the list of mappings; if not specified, an empty dict is used, so that a call to d.newchild() is equivalent to ChainMap({}, *d.maps). This method is used for creating subcontexts that can be updated without altering value in any of the parent mappings.
Changed in version 3.4:- The optional m parameter was added.
Syntax- chainmapobj.new_child(m=None)

- reversed()- This function reverse the relative ordering of dictionaries in the ChainMap.
Syntax- reversed(chainmapobj.maps)

→
```
import collections
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict3 = {'f': 5}
chain = collections.ChainMap(dict1, dict2)
print("All the ChainMap contents are:")
print(chain.maps)
chain1 = chain.new_child(dict3)
```



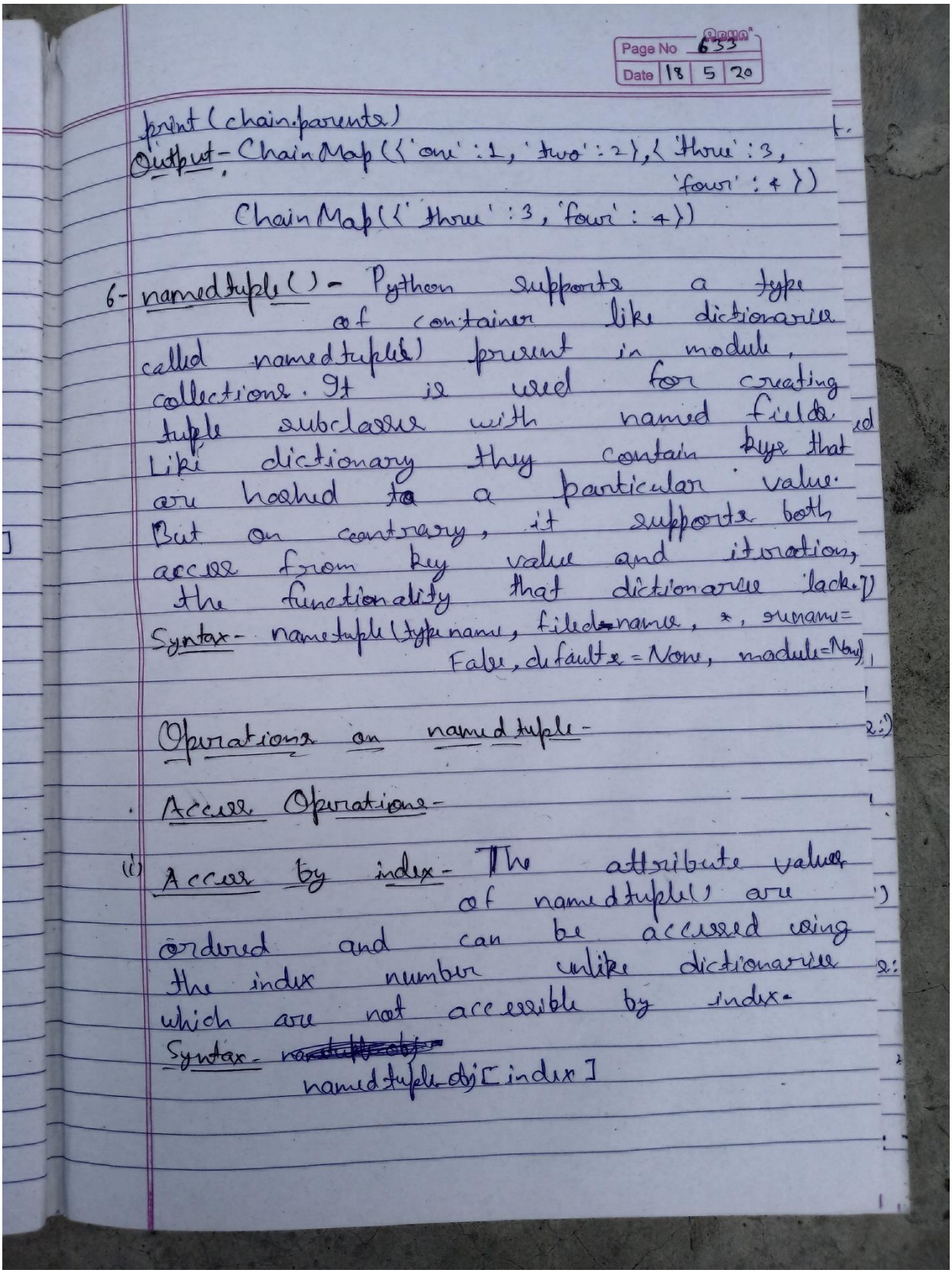
Page No. 632
Date

```
print("Displaying new ChainMap:")
print(chain1.maps)
print("Value associated with b before
                    reversing is:", end="")
print(chain['b'])
chain1.maps = reversed(chain1.maps)
print("Value associated with b after
                    reversing is:", end="")
print(chain['b'])
```

Output - All the ChainMap contents are:
[{'a': 1, 'b': 2}, {'b': 3, 'c': 4}]
Displaying new ChainMap:
[{'f': 5}, {'a': 1, 'b': 2}, {'b': 3, 'c': 4}]
Value associated with b before reversing is: 2
Value associated with b after reversing is: 3

- parents - Property returning a new ChainMap containing all of the maps in the current instance except the first one. A reference to d.parents is equivalent to: ChainMap(*d.maps[1:])
Syntax- chainmapobj.parents

→
```
from collections import ChainMap
d1={'one':1, 'two':2}
d2={'three':3, 'four':4}
chain=ChainMap(d1,d2)
print(chain)
```



Page No 655
Date 18 5 20

print(chain.parents)
Output- ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four': 4})
ChainMap({'three': 3, 'four': 4})

6- namedtuple() - Python supports a type of container like dictionaries called namedtuple() present in module, collections. It is used for creating tuple subclasses with named fields. Like dictionary they contain keys that are hashed to a particular value. But on contrary, it supports both access from key value and iteration, the functionality that dictionaries lack.

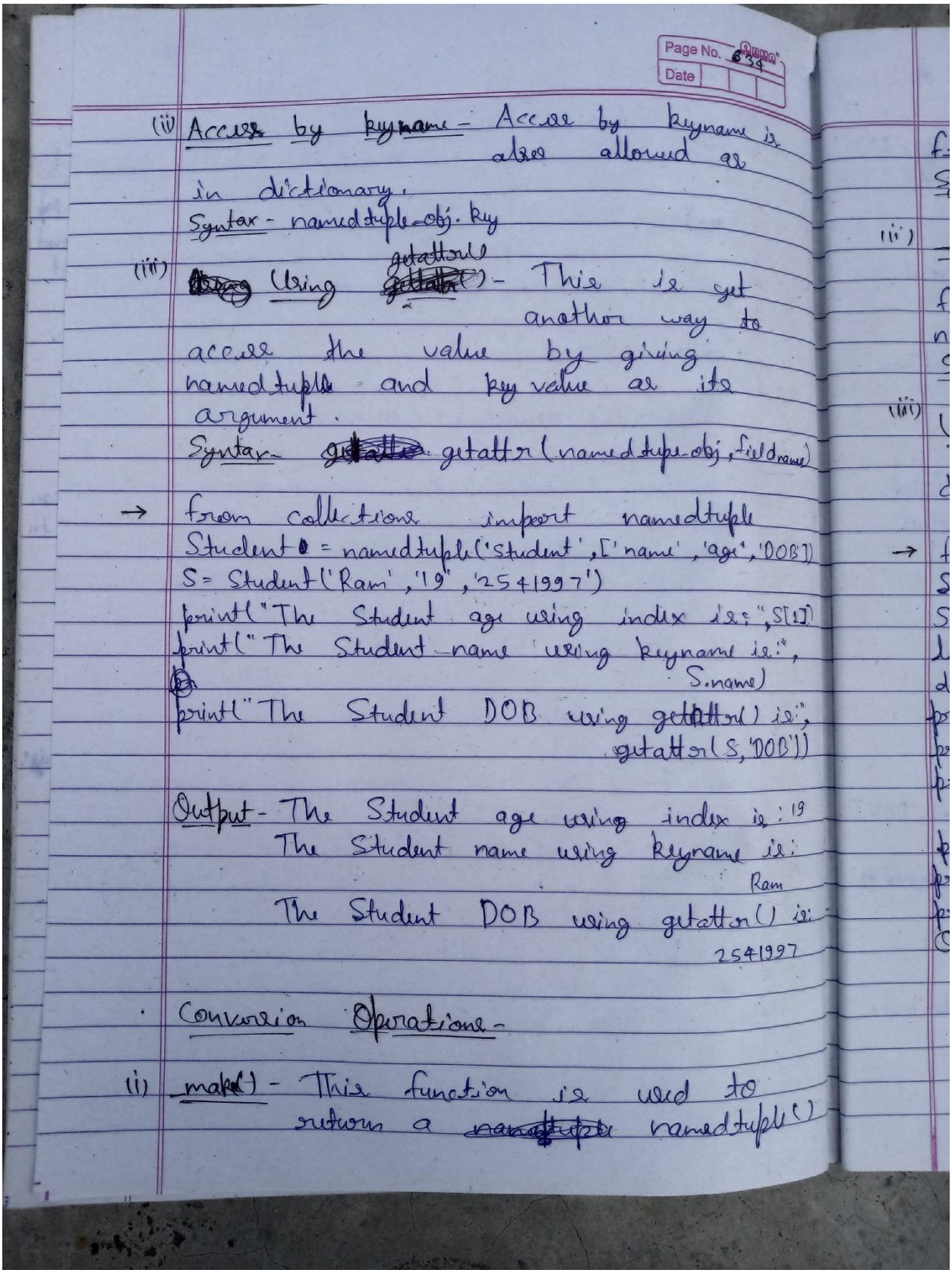
Syntax- namedtuple(typename, fieldnames, *, rename=False, defaults=None, module=None)

Operations on named tuple-

Access Operations-

(i) Access by index- The attribute values of namedtuple() are ordered and can be accessed using the index number unlike dictionaries which are not accessible by index.

Syntax- ~~namedtuple.obj~~
namedtuple_obj[index]



(ii) Access by keyname - Access by keyname is also allowed as in dictionary.
Syntax - namedtuple_obj.key

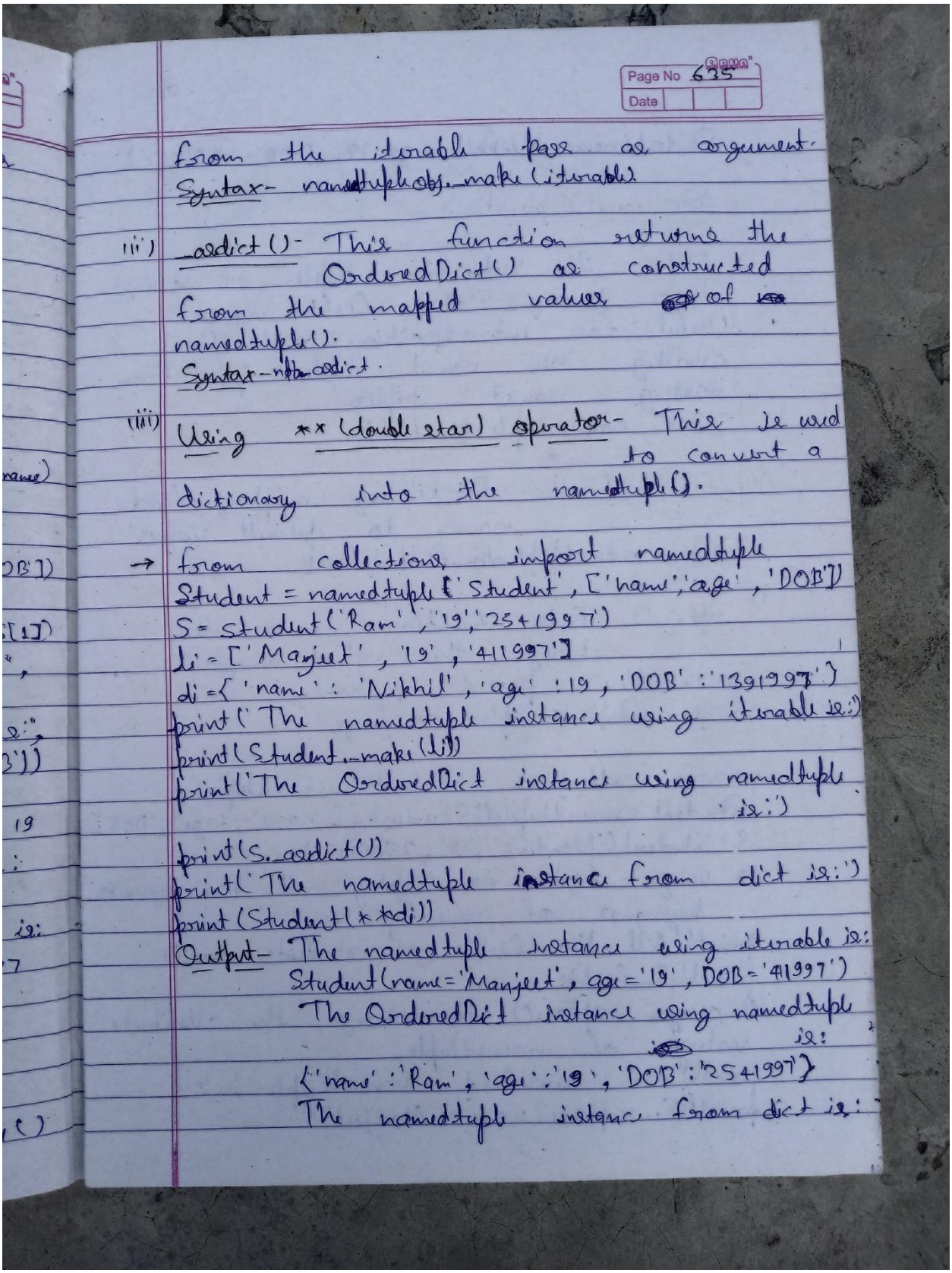
(iii) Using getattr() - This is yet another way to access the value by giving namedtuple and key value as its argument.
Syntax - getattr(namedtuple_obj, fieldname)

→
```
from collections import namedtuple
Student = namedtuple('student',['name','age','DOB'])
S = Student('Ram','19','25#1997')
print("The Student age using index is:",S[1])
print("The Student name using keyname is:",
                                    S.name)
print("The Student DOB using getattr() is:",
                        getattr(S,'DOB'))
```

Output - The Student age using index is: 19
The Student name using keyname is: Ram
The Student DOB using getattr() is: 25#1997

Conversion Operations -

(i) _make() - This function is used to return a namedtuple()



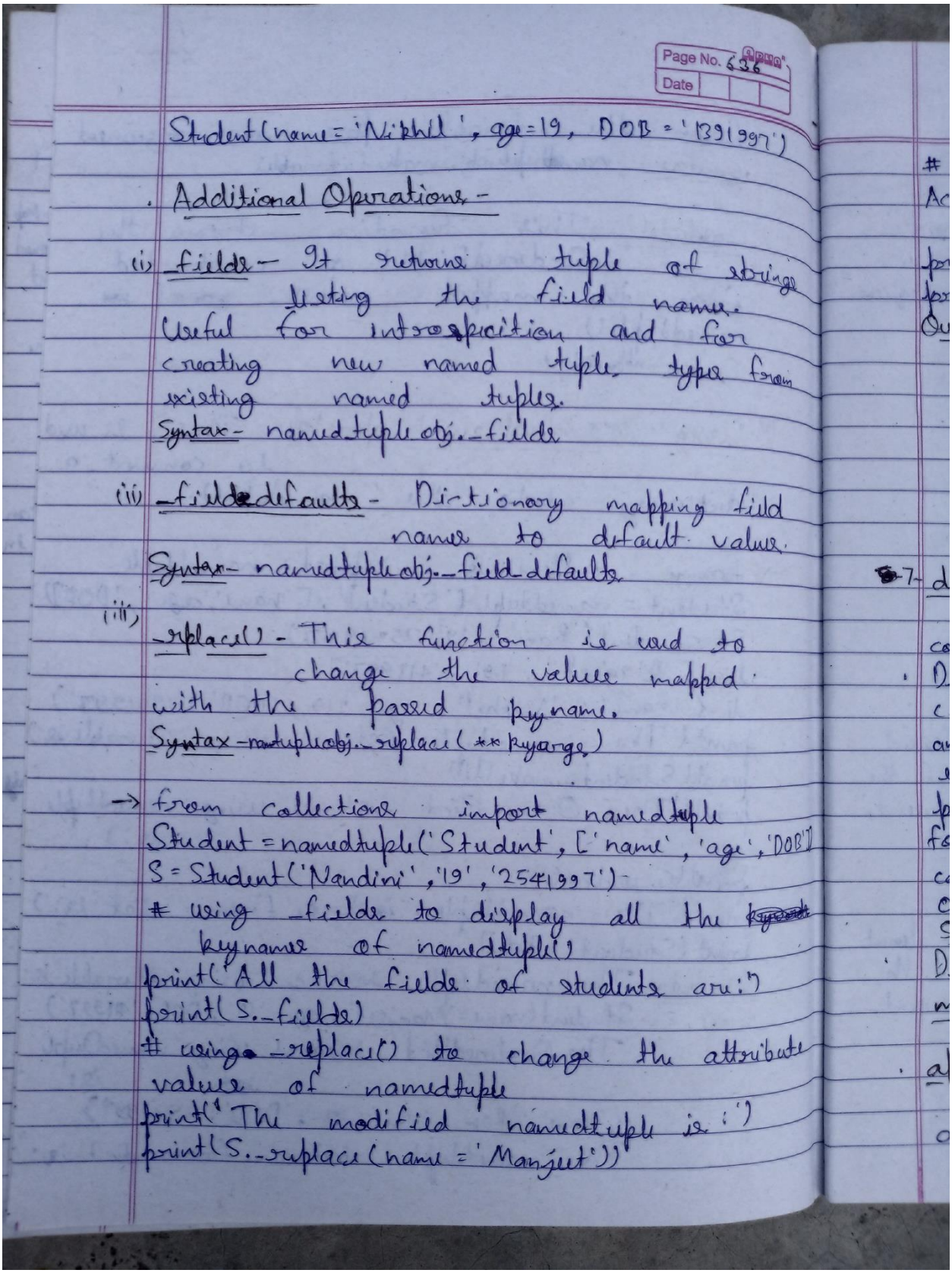
from the iterable pass as argument.
Syntax- namedtupleobj._make(iterable)

(ii) _asdict()- This function returns the OrderedDict() as constructed from the mapped values of namedtuple().
Syntax- ntba._asdict.

(iii) Using ** (double star) operator- This is used to convert a dictionary into the namedtuple().

→
```
from collections import namedtuple
Student = namedtuple('Student',['name','age','DOB'])
S = Student('Ram','19','25+1997')
li = ['Manjeet','19','411997']
di = {'name': 'Nikhil', 'age': 19, 'DOB': '1391997'}
print('The namedtuple instance using iterable is:')
print(Student._make(li))
print('The OrderedDict instance using namedtuple is:')
print(S._asdict())
print('The namedtuple instance from dict is:')
print(Student(**di))
```

Output- The namedtuple instance using iterable is:
Student(name='Manjeet', age='19', DOB='411997')
The OrderedDict instance using namedtuple is:
{'name':'Ram', 'age':'19', 'DOB':'25+1997'}
The namedtuple instance from dict is:



Student(name='Nikhil', age=19, DOB='13l1997')

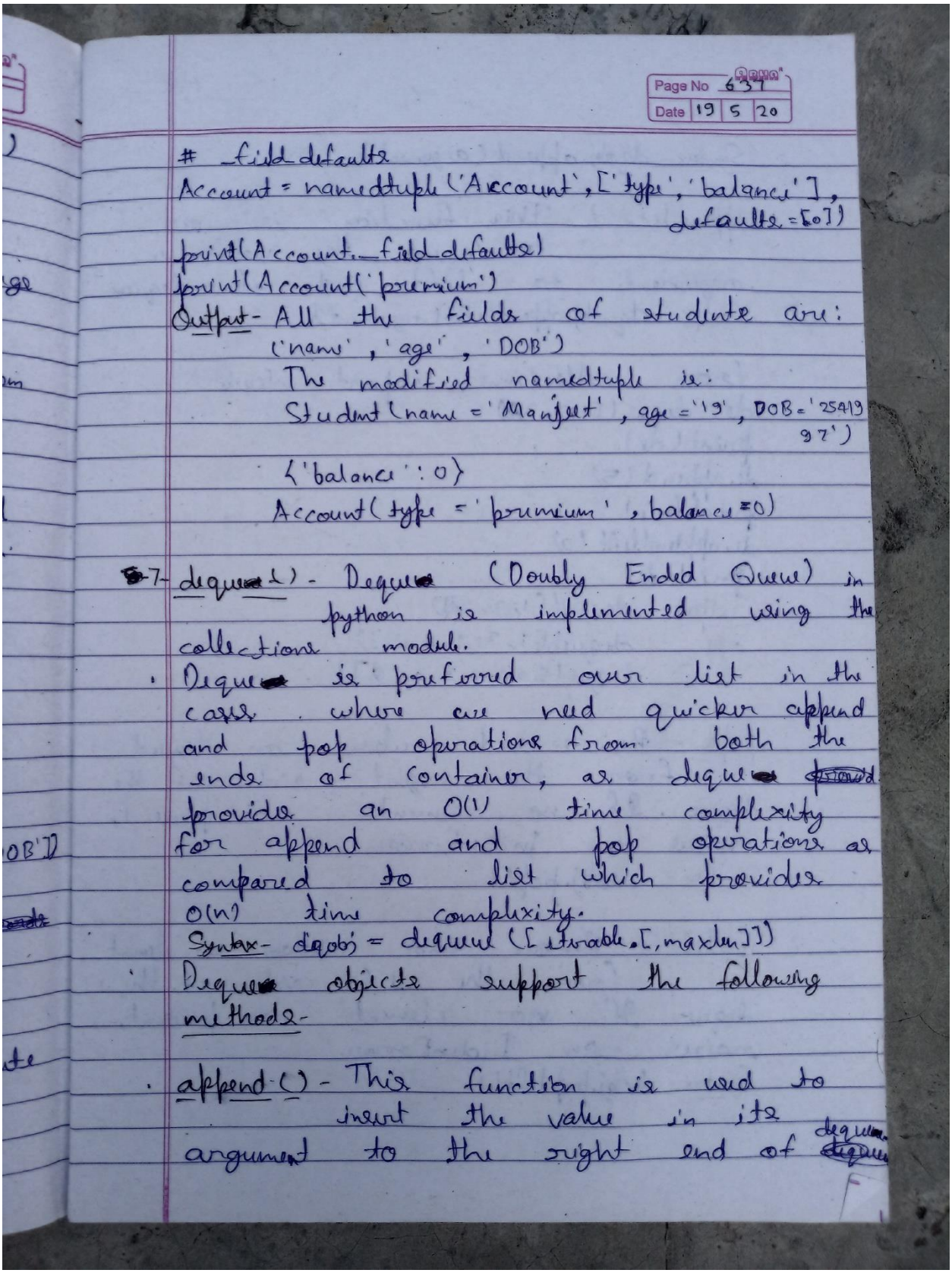
. Additional Operations -

(i) _fields - It returns tuple of strings listing the field name. Useful for introspection and for creating new named tuple type from existing named tuples.
Syntax - namedtupleobj._fields

(ii) _fields_defaults - Dictionary mapping field names to default value.
Syntax - namedtupleobj._field_defaults

(iii) _replace() - This function is used to change the values mapped with the passed keyname.
Syntax - namedtupleobj._replace(**kyargs)

→ from collections import namedtuple
Student = namedtuple('Student', ['name', 'age', 'DOB'])
S = Student('Nandini', '19', '2541997')
# using _fields to display all the keynames of namedtuple()
print('All the fields of students are:')
print(S._fields)
# using _replace() to change the attribute value of namedtuple
print('The modified namedtuple is:')
print(S._replace(name='Manjeet'))



Page No 637
Date 19 5 20

```
# field_defaults
Account = namedtuple('Account', ['type', 'balance'],
                                defaults=[0])
print(Account._field_defaults)
print(Account('premium'))
```

Output- All the fields of students are:
('name', 'age', 'DOB')
The modified namedtuple is:
Student(name='Manjeet', age='19', DOB='25/9/97')

{'balance': 0}
Account(type='premium', balance=0)

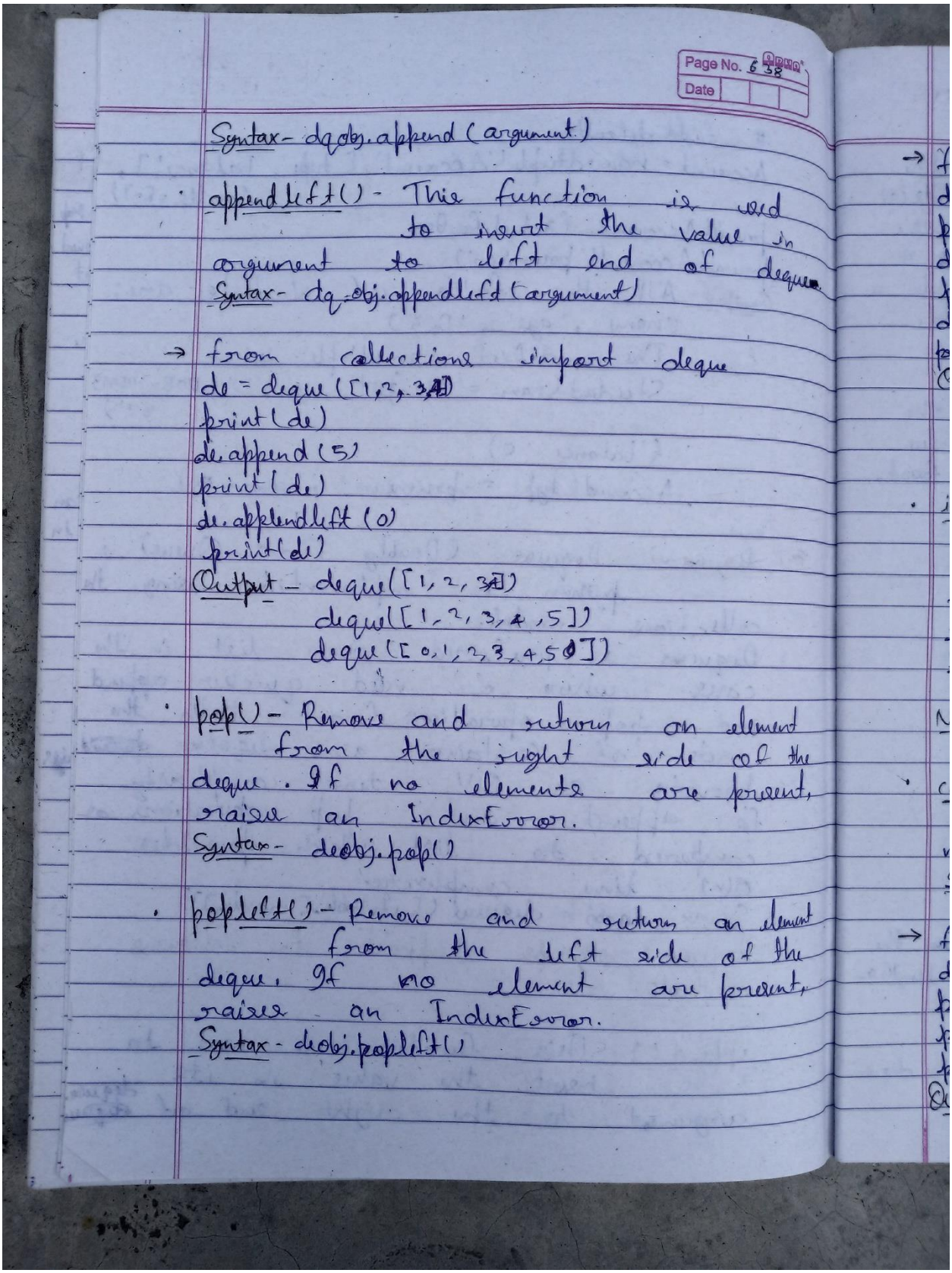
7- deque() - Deque (Doubly Ended Queue) in python is implemented using the collections module.

- Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity.

Syntax- dqobj = deque([iterable,[maxlen]])

Deque objects support the following methods-

- append() - This function is used to insert the value in its argument to the right end of deque



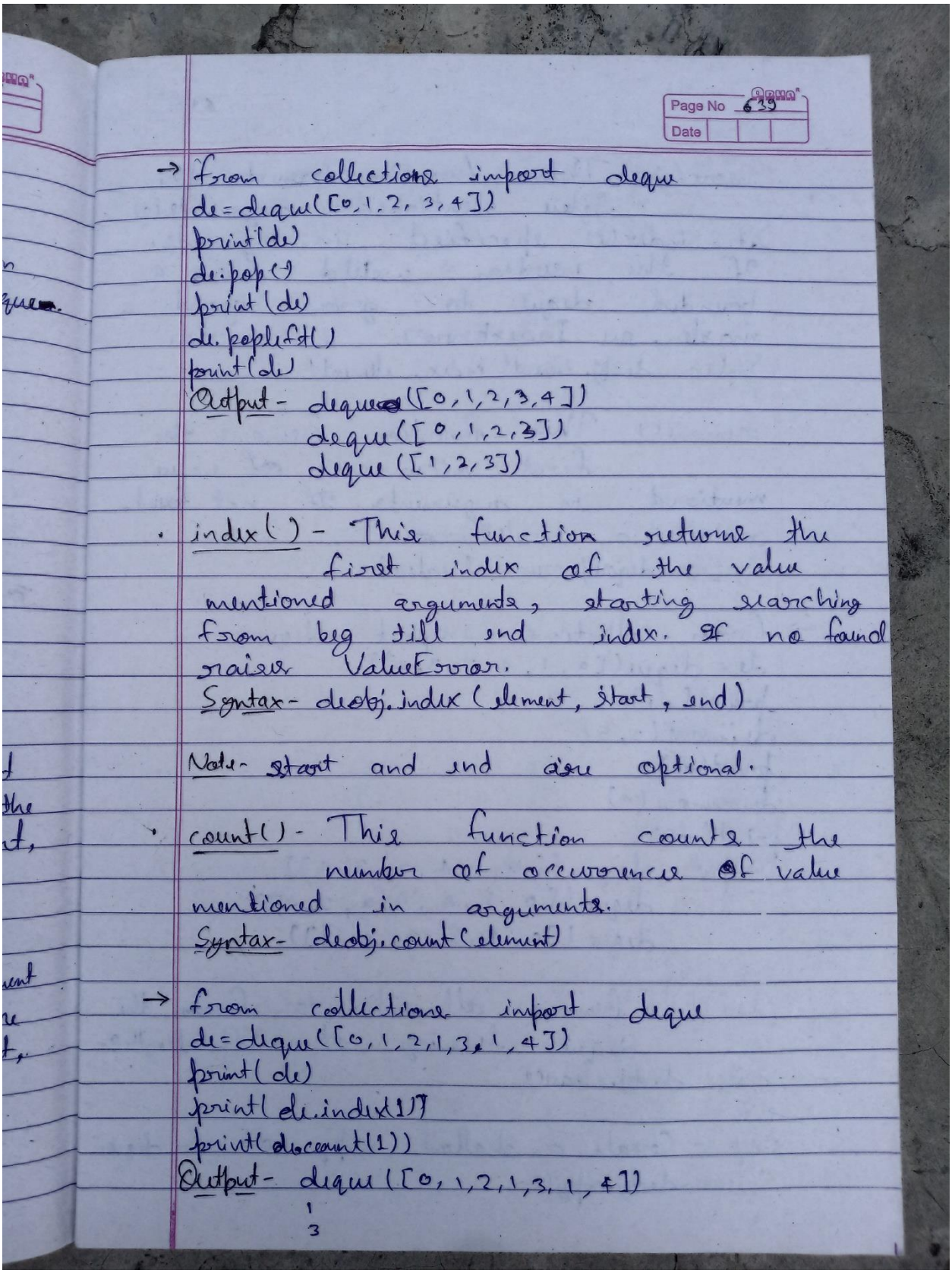
Syntax - dqobj.append(argument)

- appendleft() - This function is used to insert the value in argument to left end of deque.
  Syntax - dq_obj.appendleft(argument)

→
```
from collections import deque
de = deque([1,2,3,4])
print(de)
de.append(5)
print(de)
de.appendleft(0)
print(de)
```
Output - deque([1, 2, 3, 4])
deque([1, 2, 3, 4, 5])
deque([0, 1, 2, 3, 4, 5])

- pop() - Remove and return an element from the right side of the deque. If no elements are present, raise an IndexError.
  Syntax - deobj.pop()

- popleft() - Remove and return an element from the left side of the deque. If no element are present, raises an IndexError.
  Syntax - deobj.popleft()



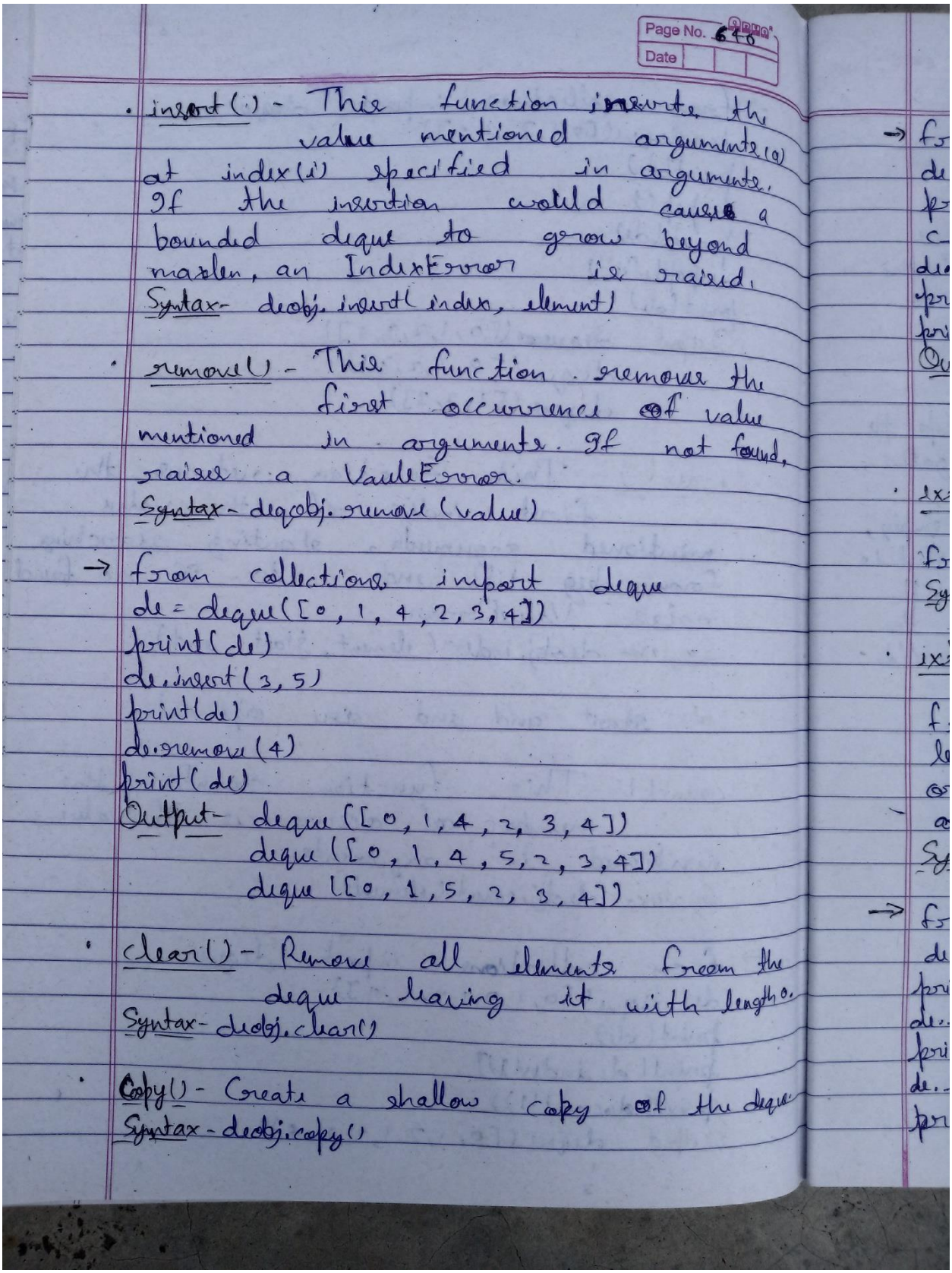
→ from collections import deque
de=deque([0,1,2,3,4])
print(de)
de.pop()
print(de)
de.popleft()
print(de)
Output- deque([0,1,2,3,4])
deque([0,1,2,3])
deque([1,2,3])

- index() - This function returns the first index of the value mentioned arguments, starting searching from beg till end index. If no found raises ValueError.
Syntax- deobj.index(element, start, end)

Note- start and end are optional.

- count() - This function counts the number of occurrences of value mentioned in arguments.
Syntax- deobj.count(element)

→ from collections import deque
de=deque([0,1,2,1,3,1,4])
print(de)
print(de.index(1))
print(de.count(1))
Output- deque([0,1,2,1,3,1,4])
1
3



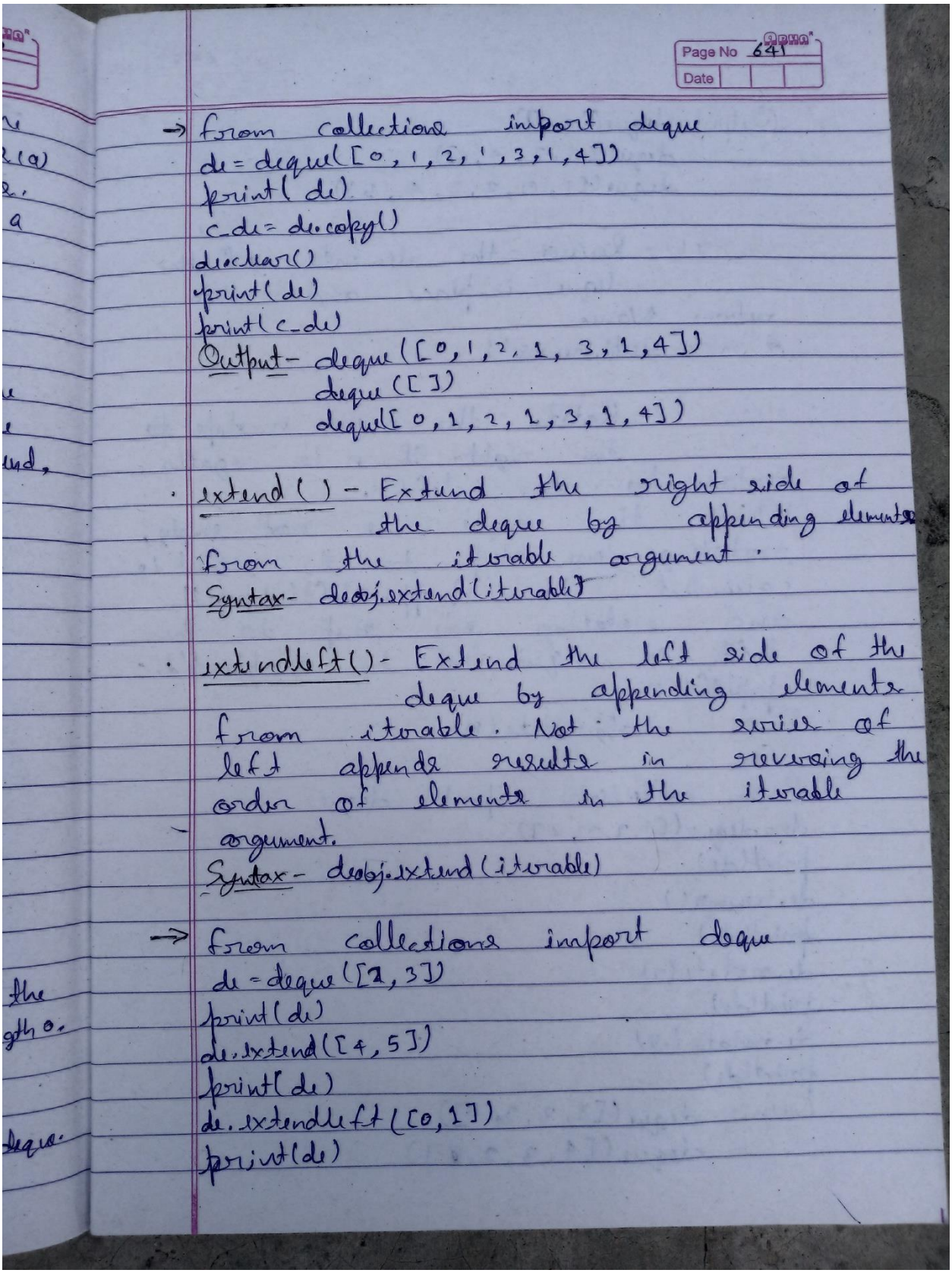
- insert() - This function inserts the value mentioned in arguments(a) at index(i) specified in arguments. If the insertion would cause a bounded deque to grow beyond maxlen, an IndexError is raised.
  Syntax- deobj.insert(index, element)

- remove() - This function removes the first occurrence of value mentioned in arguments. If not found, raises a ValueError.
  Syntax- deqobj.remove(value)

→ from collections import deque
de = deque([0, 1, 4, 2, 3, 4])
print(de)
de.insert(3, 5)
print(de)
de.remove(4)
print(de)
Output- deque([0, 1, 4, 2, 3, 4])
deque([0, 1, 4, 5, 2, 3, 4])
deque([0, 1, 5, 2, 3, 4])

- clear() - Remove all elements from the deque leaving it with length 0.
  Syntax- deobj.clear()

- Copy() - Create a shallow copy of the deque
  Syntax- deobj.copy()



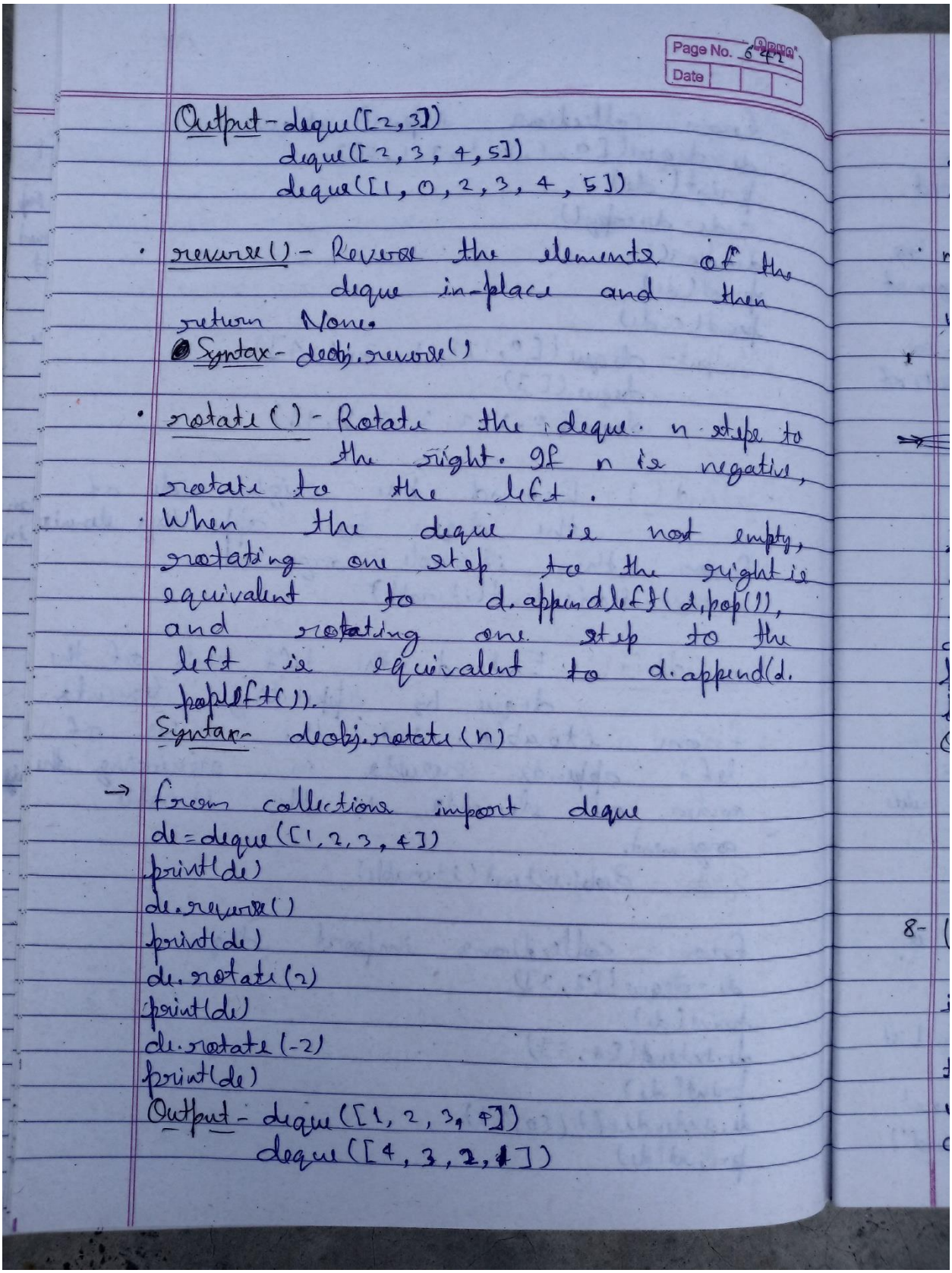
Page No 641
Date

→ from collections import deque
de = deque([0, 1, 2, 3, 1, 4])
print(de)
c_de = de.copy()
de.clear()
print(de)
print(c_de)
Output- deque([0, 1, 2, 1, 3, 1, 4])
deque([])
deque([0, 1, 2, 1, 3, 1, 4])

- extend() - Extend the right side of the deque by appending elements from the iterable argument.
Syntax- deobj.extend(iterable)

- extendleft() - Extend the left side of the deque by appending elements from iterable. Note the series of left appends results in reversing the order of elements in the iterable argument.
Syntax- deobj.extend(iterable)

→ from collections import deque
de = deque([1, 3])
print(de)
de.extend([4, 5])
print(de)
de.extendleft([0, 1])
print(de)



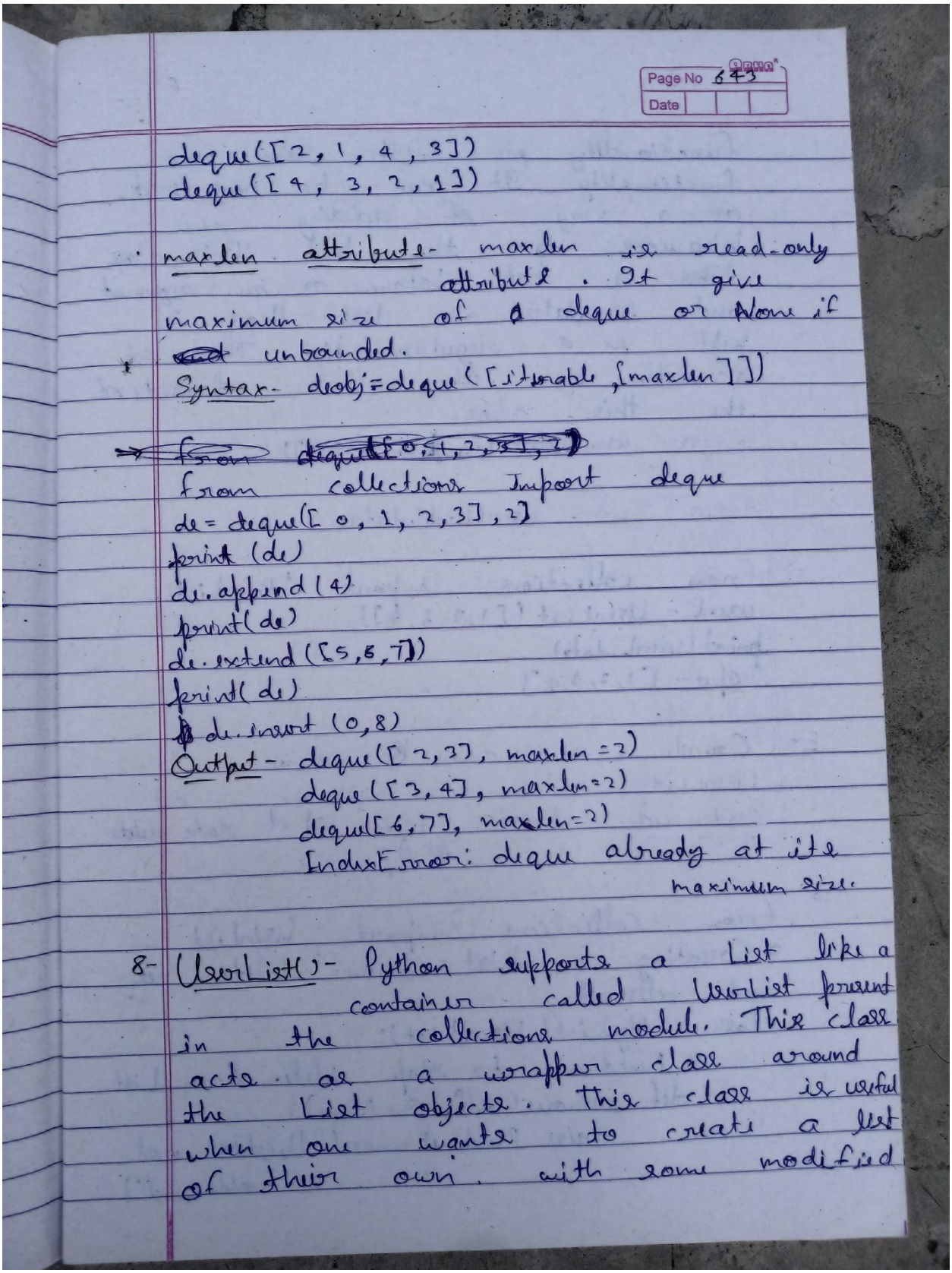
Output - deque([2,3])
deque([2,3,4,5])
deque([1,0,2,3,4,5])

- reverse() - Reverse the elements of the deque in-place and then return None.
  Syntax - deobj.reverse()

- rotate() - Rotate the deque n steps to the right. If n is negative, rotate to the left.
- When the deque is not empty, rotating one step to the right is equivalent to d.appendleft(d.pop()), and rotating one step to the left is equivalent to d.append(d.popleft()).

Syntax - deobj.rotate(n)

→
```
from collections import deque
de=deque([1,2,3,4])
print(de)
de.reverse()
print(de)
de.rotate(2)
print(de)
de.rotate(-2)
print(de)
```
Output - deque([1,2,3,4])
deque([4,3,2,1])



Page No 643
Date

deque([2, 1, 4, 3])
deque([4, 3, 2, 1])

maxlen attribute- maxlen is read-only attribute. It give maximum size of a deque or None if unbounded.

Syntax- deobj=deque([iterable,[maxlen]])

from collections import deque
de = deque([0, 1, 2, 3], 2)
print(de)
de.append(4)
print(de)
de.extend([5,6,7])
print(de)
de.insert(0,8)

Output- deque([2,3], maxlen=2)
deque([3,4], maxlen=2)
deque[6,7], maxlen=2)
IndexError: deque already at its maximum size.

8- UserList()- Python supports a List like a container called UserList present in the collections module. This class acts as a wrapper class around the List objects. This class is useful when one wants to create a list of their own with some modified

Page No. 644
Date 20 | 5 | 20

functionality or with some new functionality. It can be considered as a way of adding new behaviour for the list. This class take a list instance as an argument and simulate a list that is kept in a regular list. The list accessible by the data attribute of the this class.

Syntax - userlistobj = UserList([list])

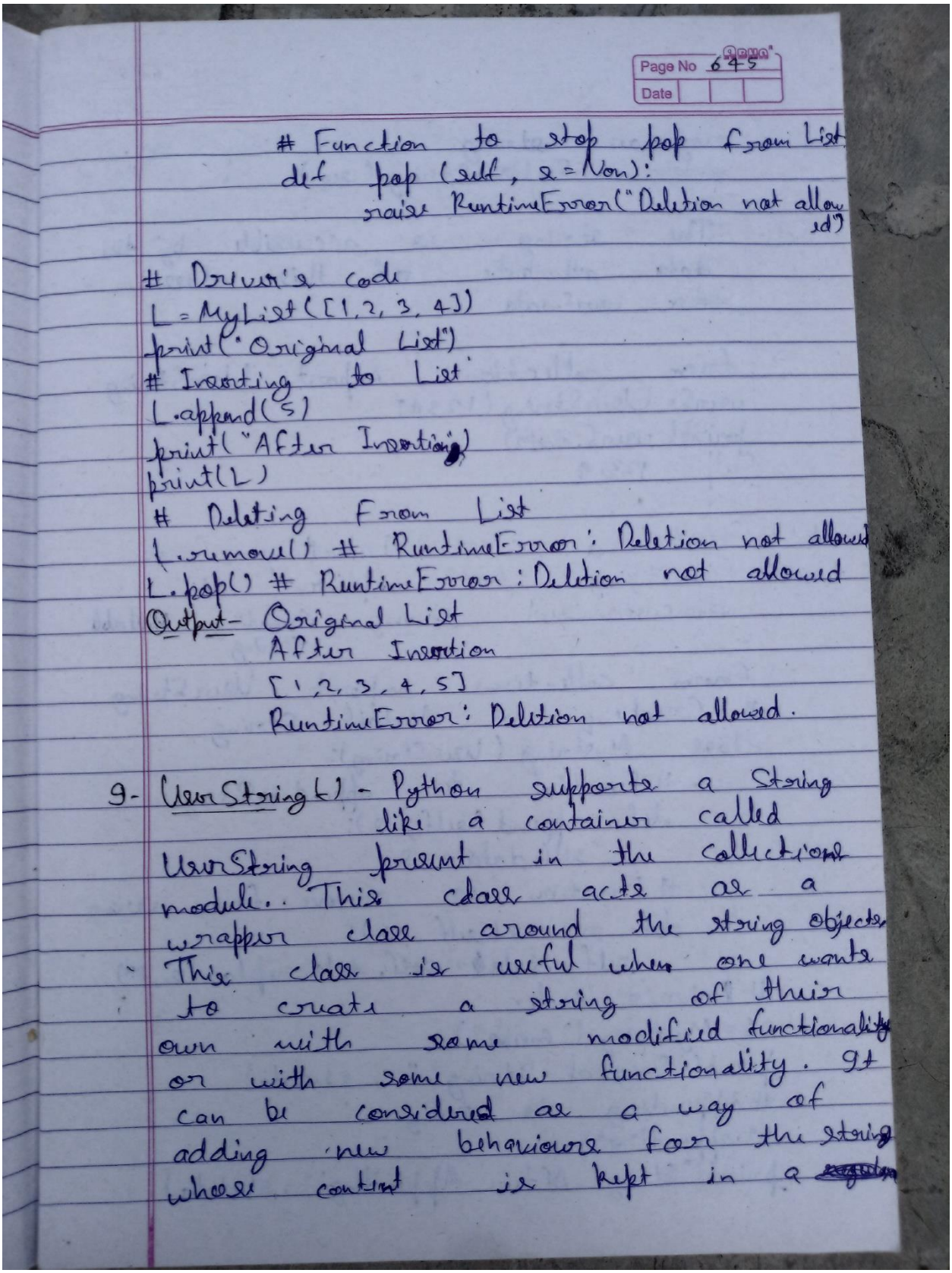
Access Syntax - userlistobj.data

→ from collections import UserList.
user = UserList([1,2,3,4])
print(user.data)
Output - [1,2,3,4]

Ex- Create a class inheriting from UserList to implement a customised List - List में से data delete न हो।

→
from collections import UserList
# Creating a List whose deletion is not allowed
class MyList(UserList):
    # Function to stop deletion from List
    def remove(self, x=None):
        raise RuntimeError("Deletion not allowed")

9-



```
# Function to stop pop from List
def pop(self, x=None):
    raise RuntimeError("Deletion not allowed")

# Driver's code
L = MyList([1, 2, 3, 4])
print("Original List")
# Inserting to List
L.append(5)
print("After Insertion")
print(L)
# Deleting From List
L.remove() # RuntimeError: Deletion not allowed
L.pop() # RuntimeError: Deletion not allowed
```

Output- Original List
After Insertion
[1, 2, 3, 4, 5]
RuntimeError: Deletion not allowed.

9- UserString() - Python supports a String like a container called UserString present in the collections module. This class acts as a wrapper class around the string objects. This class is useful when one wants to create a string of their own with some modified functionality or with some new functionality. It can be considered as a way of adding new behaviours for the string whose content is kept in a

Page No. 646
Date

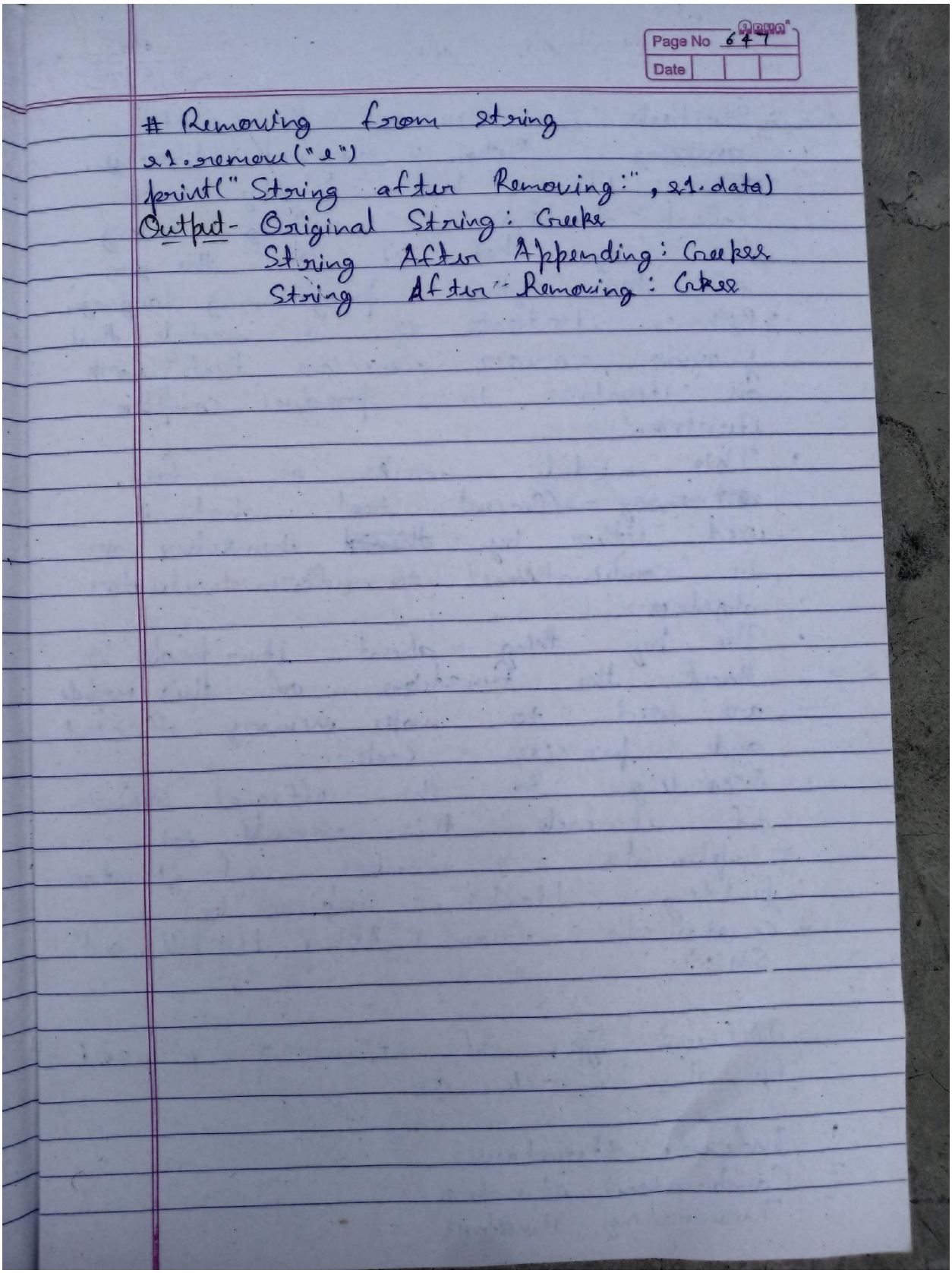
ragular string.
Syntax- userS = UserString (seq)

Note- The string is accessible by the data attribute of this class.
Syntax - userS.data

→ from collections import UserString
userS = UserString (1234)
print(userS.data)
Output- 1234

Ex- Create a class Inheriting from UserString to implement. A customised String - Creating Mutable String

→ from collections import UserString
# Creating a Mutable String
class Mystring (UserString):
    # Function to append toString
    def append (self, s):
        self.data += s
    # Function to remove from string
    def remove (self, s):
        self.data = self.data.replace(s, "")
# Driver's code.
s1 = Mystring ("Geeks")
print("Original String:", s1.data)
# Appending to string
s1.append("s")
print("String After Appending:", s1.data)



Page No 647
Date

# Removing from string
s1.remove("e")
print("String after Removing:", s1.data)
Output- Original String: Geeks
String After Appending: Geekss
String After Removing: Gkss