Python Itertools Module

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

SATYAM SETH

22-09-2021

Itertools Module

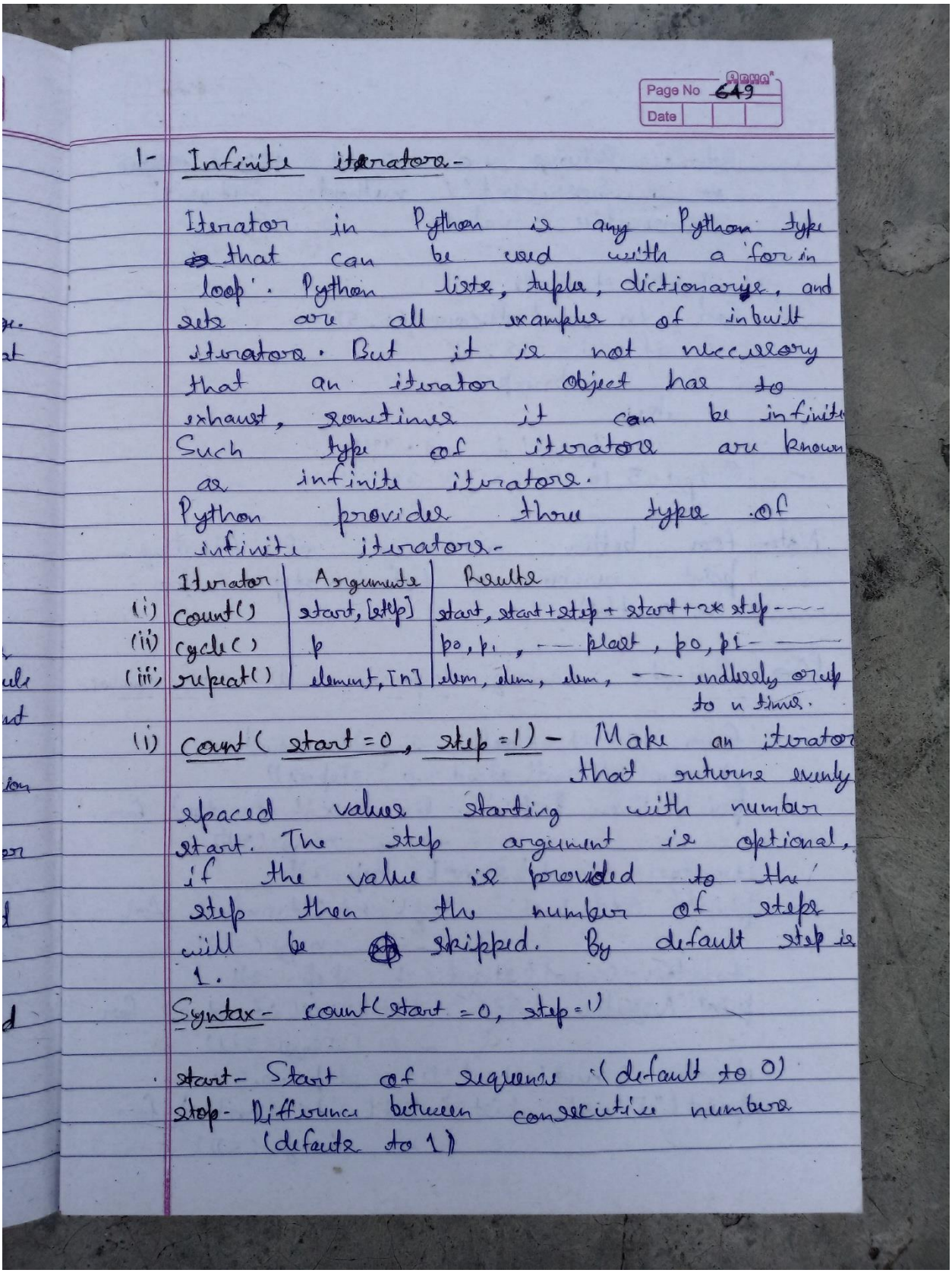
Page No. 648
Date 21 5 20

1-

- Itertools is one of the most amazing Python 3 standard libraries.
- This library has pretty much collect functions and nothing wrong to say that it is the gem of the Python programming language.
- Python's itertools is a module that provide various functions that work on iterators to produce complex iterators.
- This module works as a fast, memory-efficient tool that is used either by themselves or in combination to form iterator algebra (i)
- The key thing about itertools is (ii)
that the functions of this module (iii)
are used to make memory-efficient and precise code. (i)
- According to the official defination of itertools " this module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML.

Different types of iterators provided by this module are-

1- Infinite iterators
2- Combinatoric iterators
3- Terminating iterators



Page No 649
Date

1- Infinite iterators-

Iterator in Python is any Python type that can be used with a 'for in loop'. Python lists, tuples, dictionaries, and sets are all examples of inbuilt iterators. But it is not necessary that an iterator object has to exhaust, sometimes it can be infinite. Such type of iterators are known as infinite iterators.

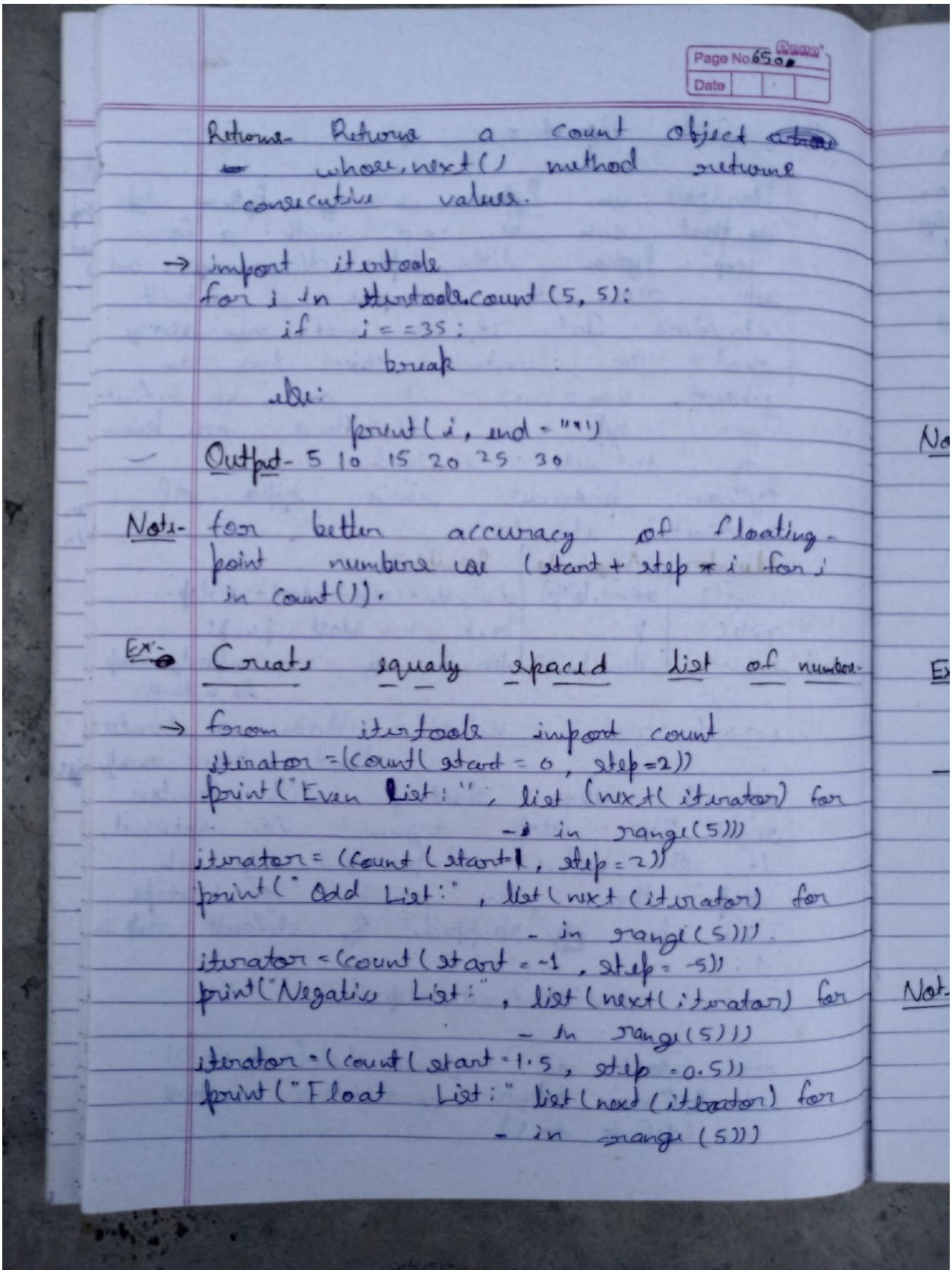
Python provides three types of infinite iterators-

	Iterator	Arguments	Results
(i)	count()	start, [step]	start, start+step, start+2*step, ---
(ii)	cycle()	p	p0, p1, --- plast, p0, p1 ---
(iii)	repeat()	element, [n]	elem, elem, elem, --- endlessly or up to n times.

(i) count(start=0, step=1) - Make an iterator that returns evenly spaced values starting with number start. The step argument is optional, if the value is provided to the step then the number of steps will be skipped. By default step is 1.

Syntax- count(start=0, step=1)

start- Start of sequence (default to 0)
step- Difference between consecutive numbers (defaults to 1)



Page No 65
Date

Returns- Returns a count object whose .next() method returns consecutive values.

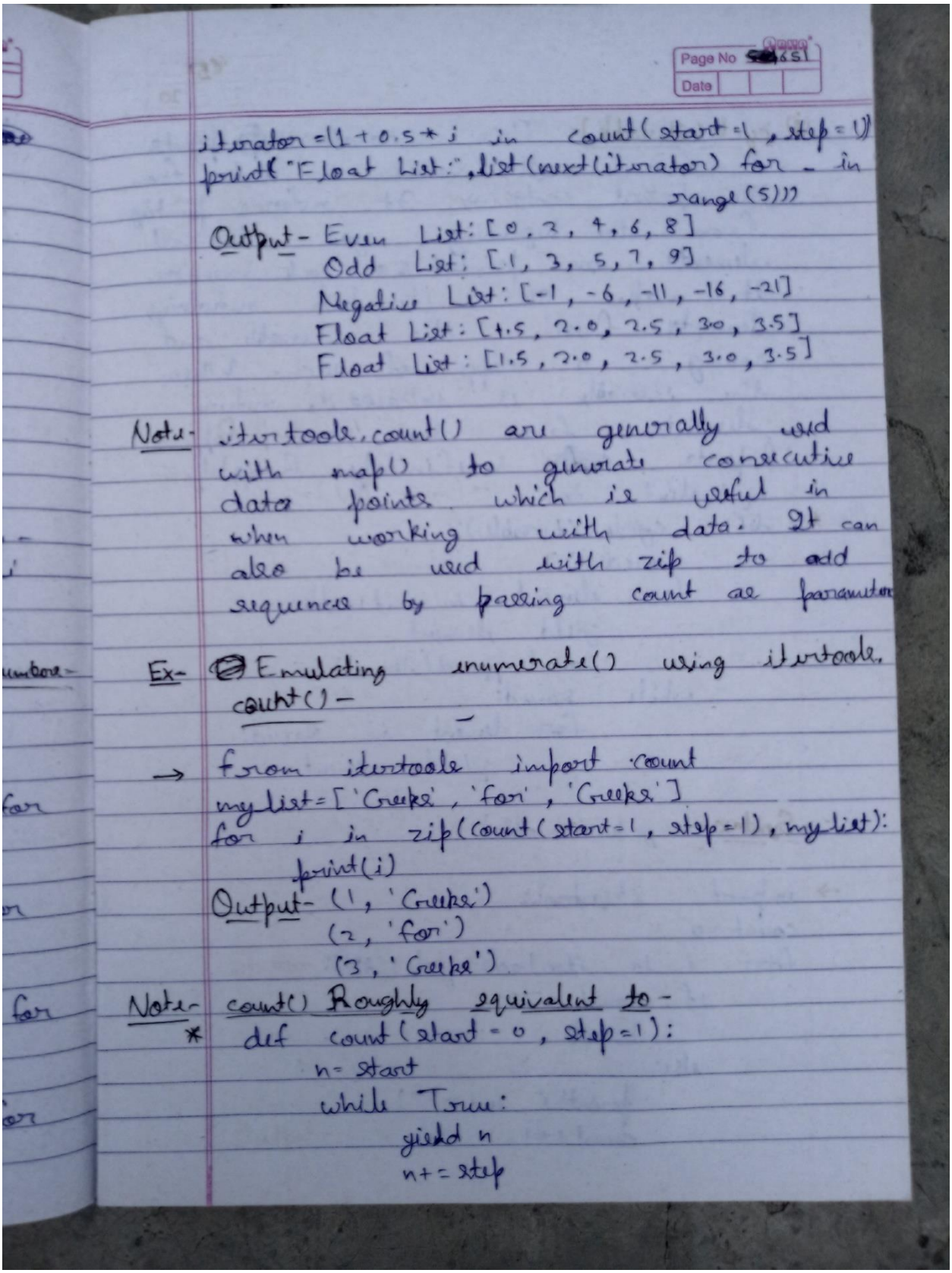
→ import itertools
for i in itertools.count(5, 5):
    if i == 35:
        break
    else:
        print(i, end = " ")

Output- 5 10 15 20 25 30

Note- for better accuracy of floating-point numbers use (start + step * i for i in count()).

Ex- Create equally spaced list of numbers.

→ from itertools import count
iterator = (count(start = 0, step = 2))
print("Even List:", list (next(iterator) for
    - in range(5)))
iterator = (count (start = 1, step = 2))
print(" Odd List:", list (next (iterator) for
    - in range(5)))
iterator = (count (start = -1, step = -5))
print("Negative List:", list (next(iterator) for
    - in range(5)))
iterator = (count (start = 1.5, step = 0.5))
print("Float List:" list (next (iterator) for
    - in range (5)))



```
iterator = (1 + 0.5 * i for i in count(start=1, step=1))
print("Float List:", list(next(iterator) for _ in
                                    range(5)))
```

Output- Even List: [0, 2, 4, 6, 8]
Odd List: [1, 3, 5, 7, 9]
Negative List: [-1, -6, -11, -16, -21]
Float List: [1.5, 2.0, 2.5, 3.0, 3.5]
Float List: [1.5, 2.0, 2.5, 3.0, 3.5]

Note- itertools.count() are generally used with map() to generate consecutive data points which is useful in when working with data. It can also be used with zip to add sequence by passing count as parameter

Ex- Emulating enumerate() using itertools.count() -

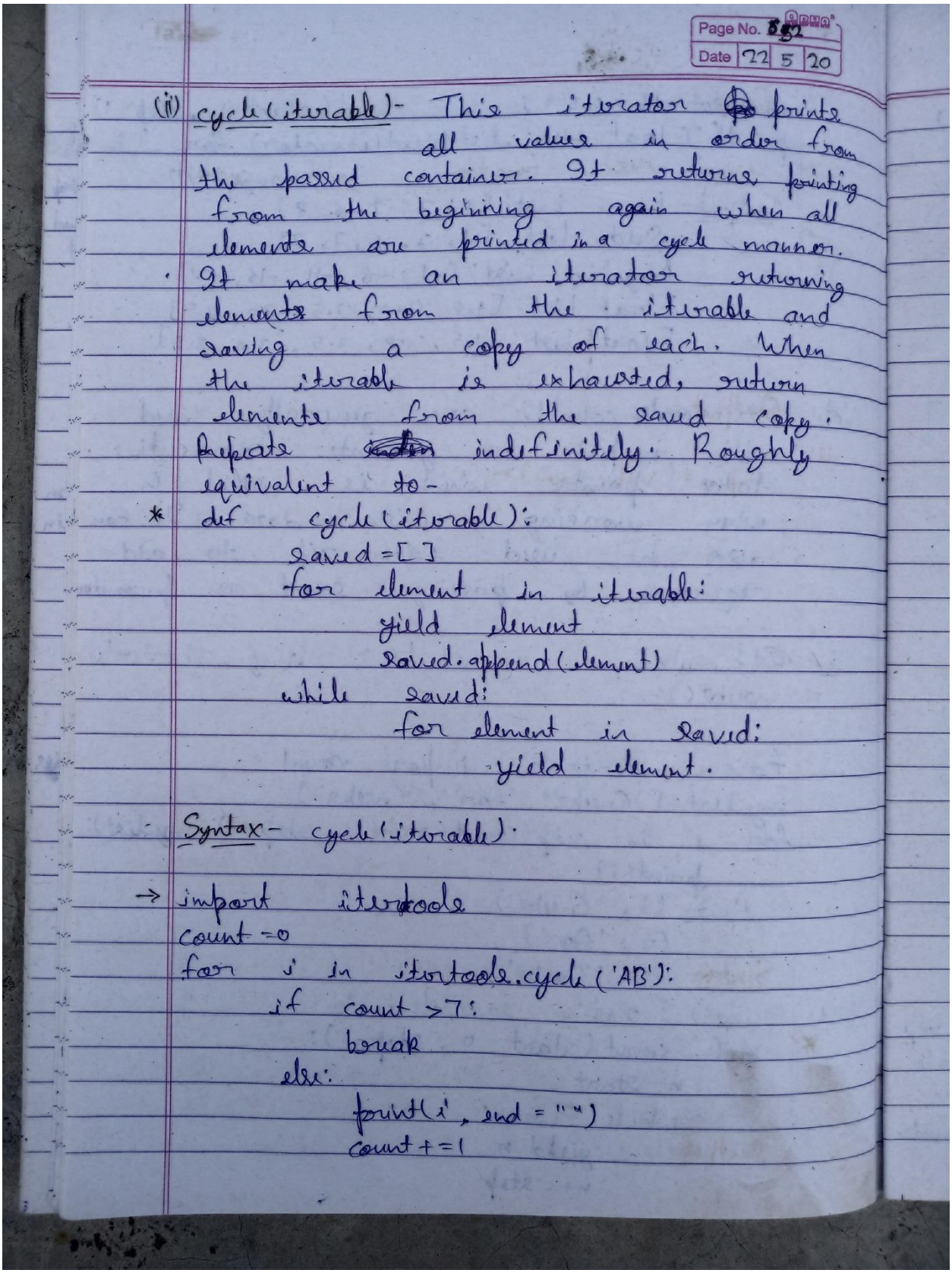
→

```
from itertools import count
my_list = ['Geeks', 'for', 'Geeks']
for i in zip(count(start=1, step=1), my_list):
    print(i)
```

Output- (1, 'Geeks')
(2, 'for')
(3, 'Geeks')

Note- * count() Roughly equivalent to -

```
def count(start=0, step=1):
    n = start
    while True:
        yield n
        n += step
```

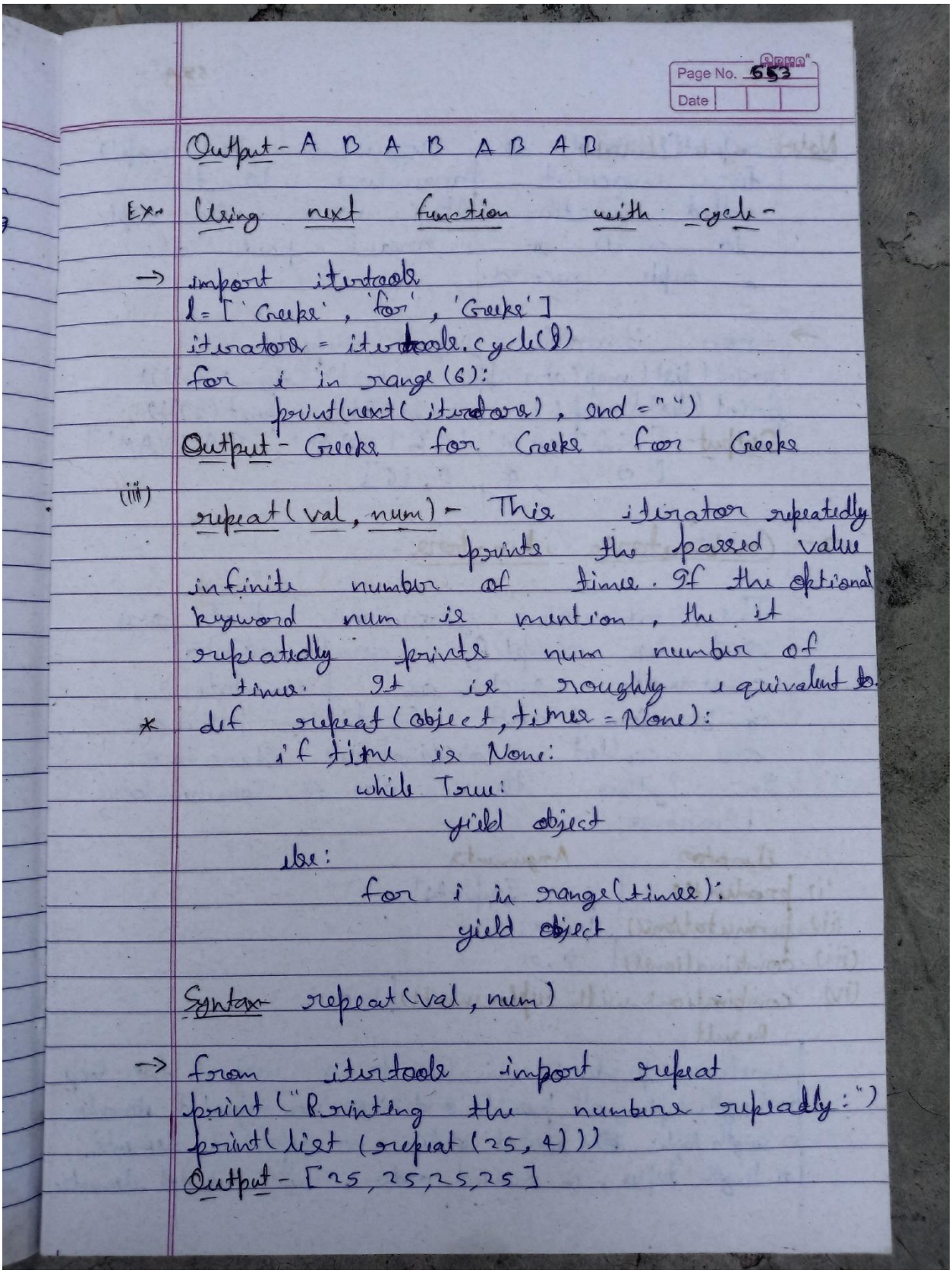


(ii) cycle(iterable)- This iterator prints all values in order from the passed container. It returns printing from the beginning again when all elements are printed in a cycle manner. It make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to-

```
* def cycle(iterable):
      saved=[]
      for element in iterable:
          yield element
          saved.append(element)
      while saved:
          for element in saved:
              yield element.
```

Syntax- cycle(iterable).

→
```
import itertools
count=0
for i in itertools.cycle('AB'):
    if count>7:
        break
    else:
        print(i, end="")
        count+=1
```



Page No. 653
Date

Output - A B A B A B A B

Ex. Using next function with cycle -

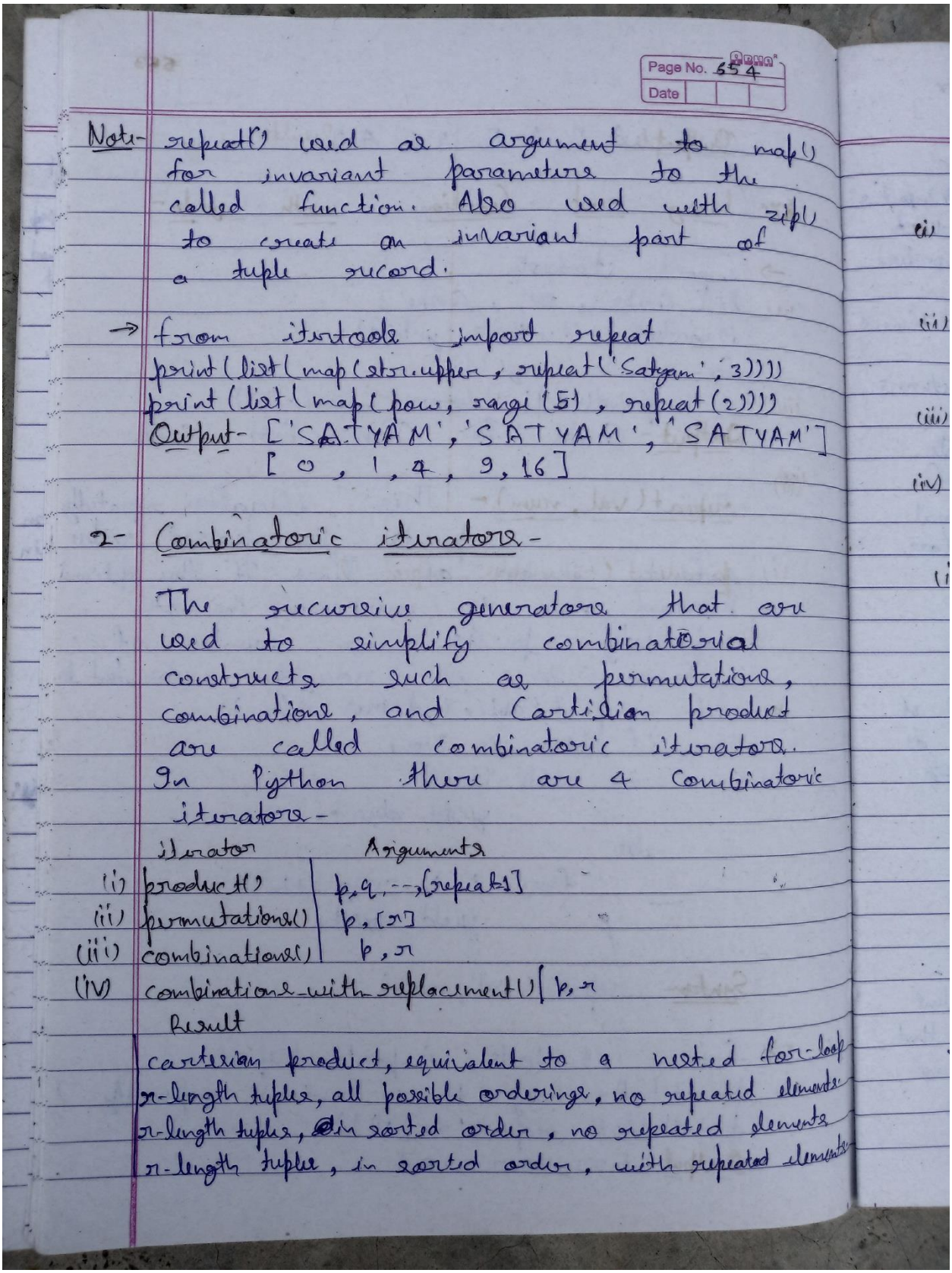
→
```
import itertools
l = ['Geeks', 'for', 'Geeks']
iterators = itertools.cycle(l)
for i in range(6):
    print(next(iterators), end = " ")
```
Output - Geeks for Geeks for Geeks

(iii) repeat(val, num) - This iterator repeatedly prints the passed value infinite number of time. If the optional keyword num is mention, the it repeatedly prints num number of time. It is roughly equivalent to.

*
```
def repeat(object, times = None):
    if time is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

Syntax repeat(val, num)

→
```
from itertools import repeat
print("Printing the numbers repeatedly:")
print(list(repeat(25, 4)))
```
Output - [25, 25, 25, 25]



Page No. 554
Date

Note- repeat() used as argument to map() for invariant parameters to the called function. Also used with zip() to create an invariant part of a tuple record.

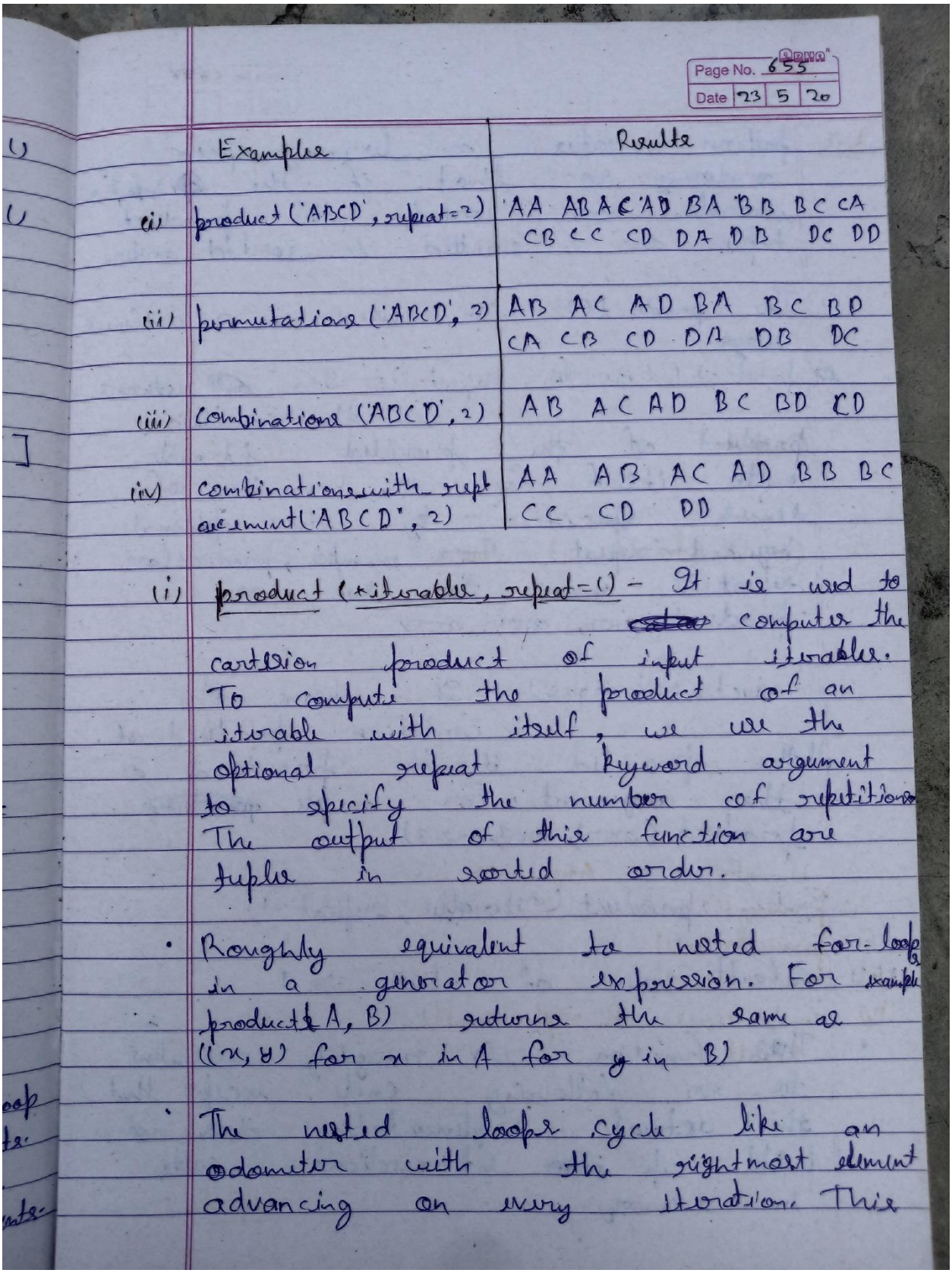
→ from itertools import repeat
print(list(map(str.upper, repeat('Satyam', 3))))
print(list(map(pow, range(5), repeat(2))))
Output- ['SATYAM', 'SATYAM', 'SATYAM']
[0, 1, 4, 9, 16]

2- Combinatoric iterators-

The recursive generators that are used to simplify combinatorial constructs such as permutations, combinations, and Cartesian product are called combinatoric iterators.
In Python there are 4 combinatoric iterators-

	Iterator	Arguments
(i)	product()	p, q, ..., [repeat=1]
(ii)	permutations()	p, [r]
(iii)	combinations()	p, r
(iv)	combinations_with_replacement()	p, r

Result
cartesian product, equivalent to a nested for-loop
r-length tuples, all possible orderings, no repeated elements
r-length tuples, in sorted order, no repeated elements
r-length tuples, in sorted order, with repeated elements



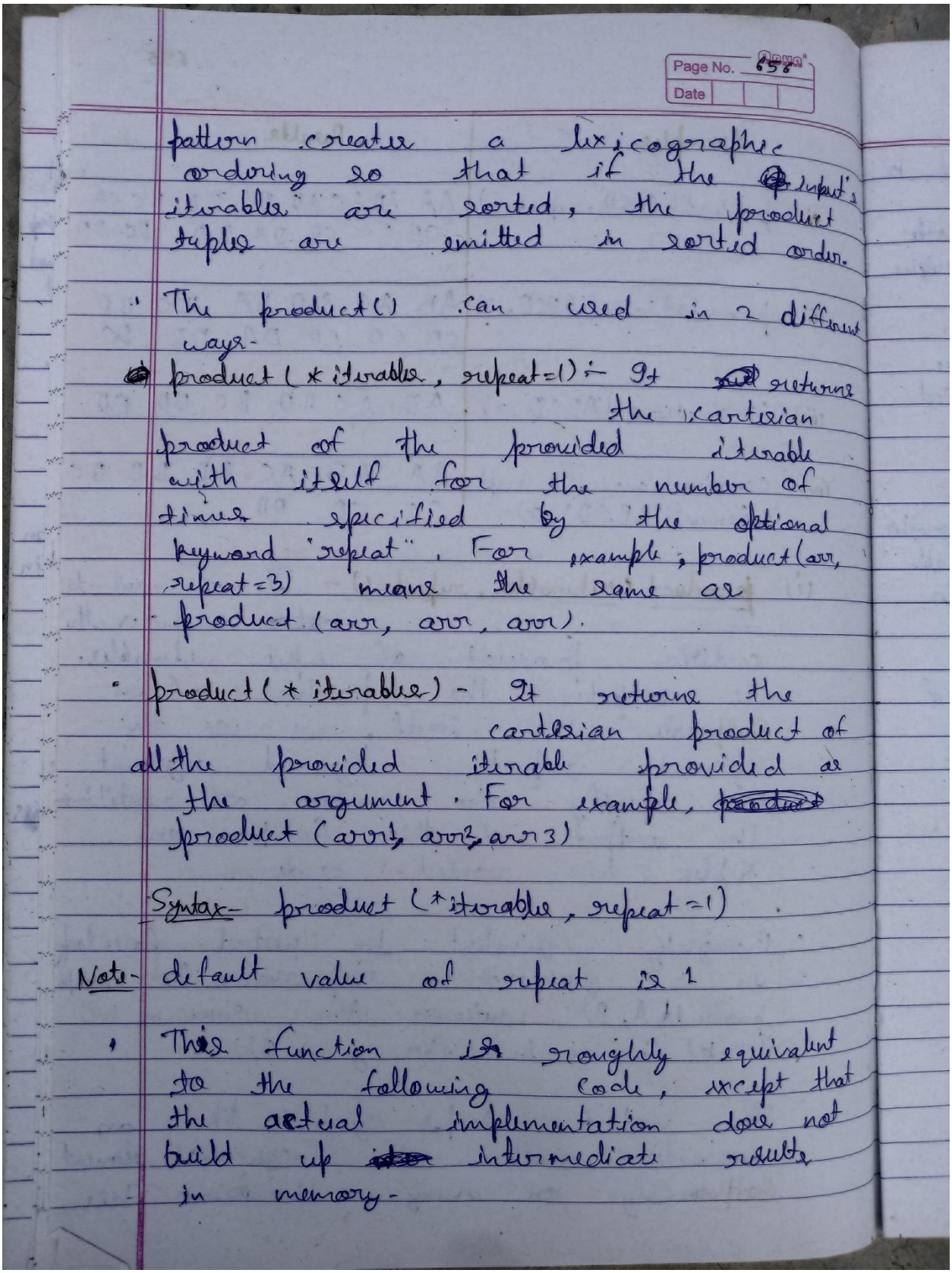
Page No. 655
Date 23 5 20

	Examples	Results
(i)	product('ABCD', repeat=2)	AA AB AC AD BA BB BC CA CB CC CD DA DB DC DD
(ii)	permutations('ABCD', 2)	AB AC AD BA BC BD CA CB CD DA DB DC
(iii)	combinations('ABCD', 2)	AB AC AD BC BD CD
(iv)	combinations_with_replacement('ABCD', 2)	AA AB AC AD BB BC CC CD DD

(i) product(*iterable, repeat=1) - It is used to compute the cartesian product of input iterables. To compute the product of an iterable with itself, we use the optional repeat keyword argument to specify the number of repetitions. The output of this function are tuple in sorted order.

- Roughly equivalent to nested for-loops in a generator expression. For example product(A, B) returns the same as ((x, y) for x in A for y in B)

- The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This



Page No. 658
Date

pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

- The product() can used in 2 different ways-

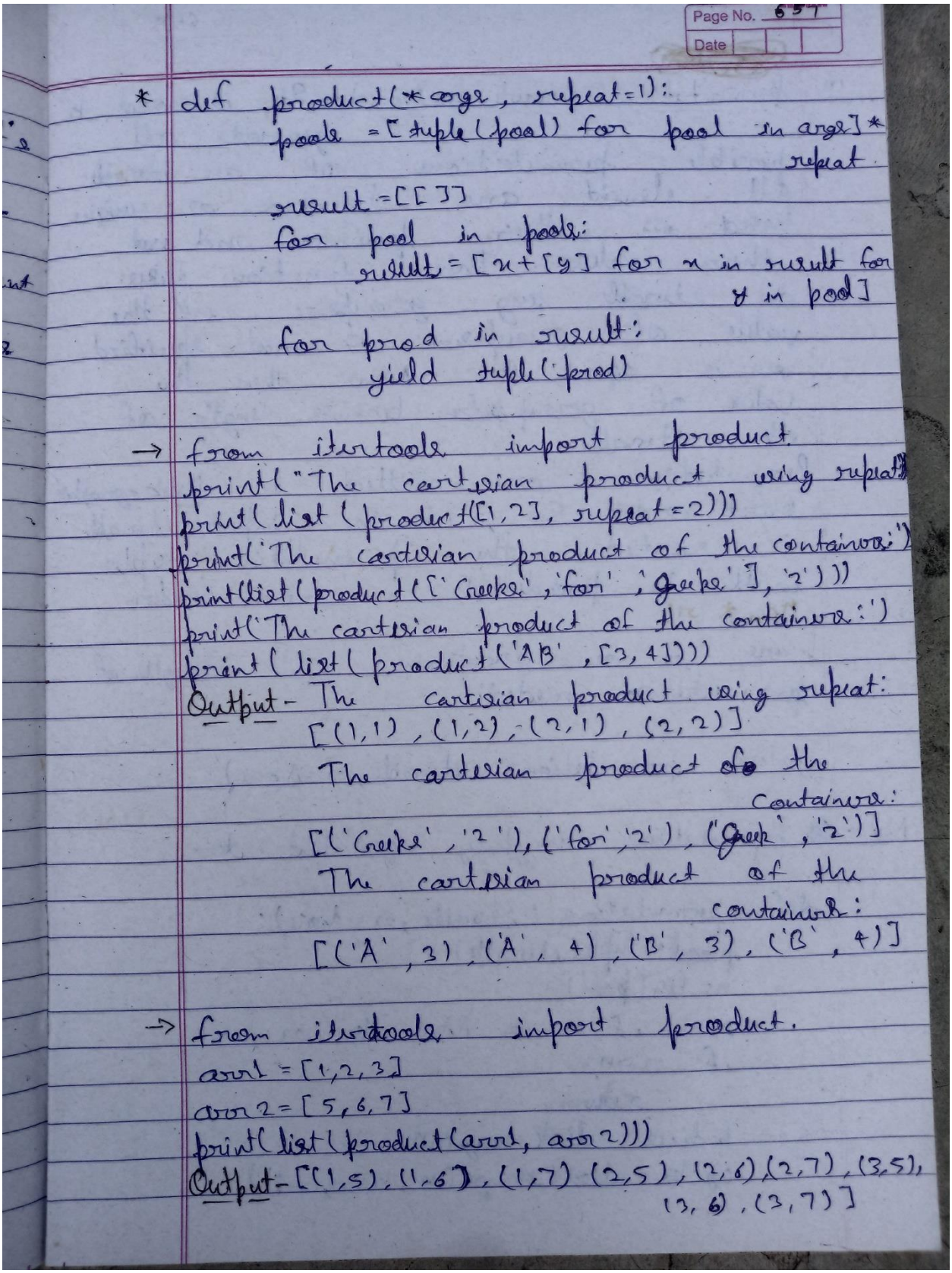
- product(*iterable, repeat=1) :- It returns the cartesian product of the provided iterable with itself for the number of times specified by the optional keyword "repeat". For example, product(arr, repeat=3) means the same as product(arr, arr, arr).

- product(*iterable) - It returns the cartesian product of all the provided iterable provided as the argument. For example, product(arr1, arr2, arr3)

Syntax- product(*iterable, repeat=1)

Note- default value of repeat is 1

- This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory-



Page No. 657
Date

* def product(*args, repeat=1):
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)

→ from itertools import product
print("The cartesian product using repeat:")
print(list(product([1,2], repeat=2)))
print("The cartesian product of the containers:")
print(list(product(['Geeks', 'for', 'Geeks'], '2')))
print("The cartesian product of the containers:")
print(list(product('AB', [3,4])))
Output- The cartesian product using repeat:
[(1,1), (1,2), (2,1), (2,2)]
The cartesian product of the containers:
[('Geeks', '2'), ('for', '2'), ('Geek', '2')]
The cartesian product of the containers:
[('A', 3), ('A', 4), ('B', 3), ('B', 4)]

→ from itertools import product.
arr1 = [1,2,3]
arr2 = [5,6,7]
print(list(product(arr1, arr2)))
Output- [(1,5), (1,6), (1,7) (2,5), (2,6), (2,7), (3,5), (3,6), (3,7)]

Page No. 658
Date

(ii) permutations(iterable, r=None) - It is used to generate all possible permutations of an iterable. All element are treated as unique based on their position and not their values. This function take an iterable and groupsize, if the value of group-size is not specified or is equal to None then the value of group-size become length of the iterable.

- Permutations are emitted in lexicographic sort order. So, if the input iterable is sorted, the permutation tuple will be produced in sorted order.

$^nC_r * r!$

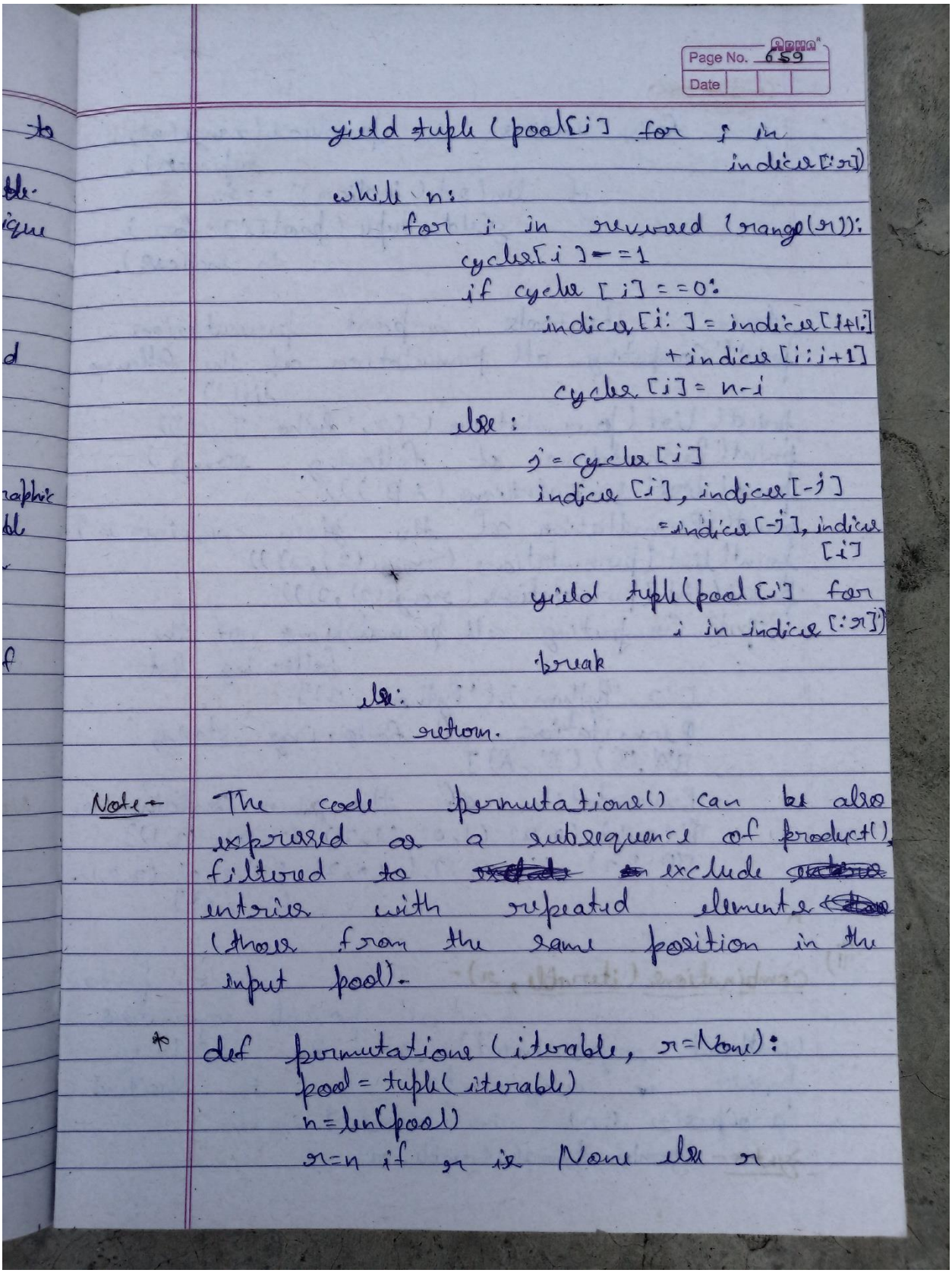
Where r or group-size is length of permutation needed).

Syntax- permutations(iterable, r=None).

Note- permutations roughly equivalent to-

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
```

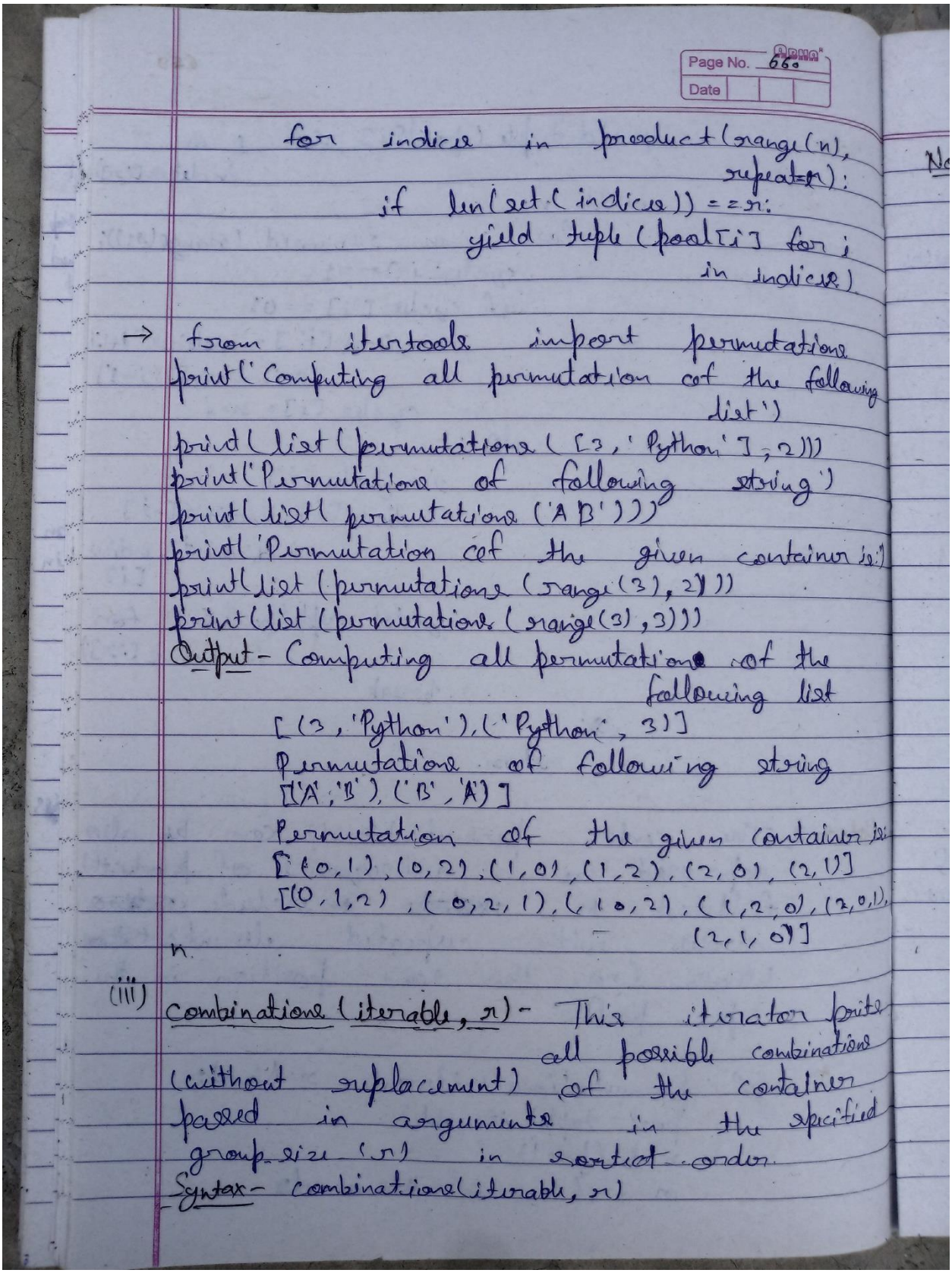
Note



```
        yield tuple(pool[i] for i in
                                indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:]
                            + indices[i:i+1]
                cycles[i] = n-i
            else:
                j = cycles[i]
                indices[i], indices[-j]
                    = indices[-j], indices
                                    [i]
                yield tuple(pool[i] for
                        i in indices[:r])
                break
        else:
            return.
```

Note:- The code permutations() can be also expressed as a subsequence of product(), filtered to ~~exclude~~ exclude ~~entries~~ entries with repeated elements ~~(those~~ (those from the same position in the input pool).

```
* def permutations(iterable, r=None):
      pool = tuple(iterable)
      n = len(pool)
      r = n if r is None else r
```



Page No. 66
Date

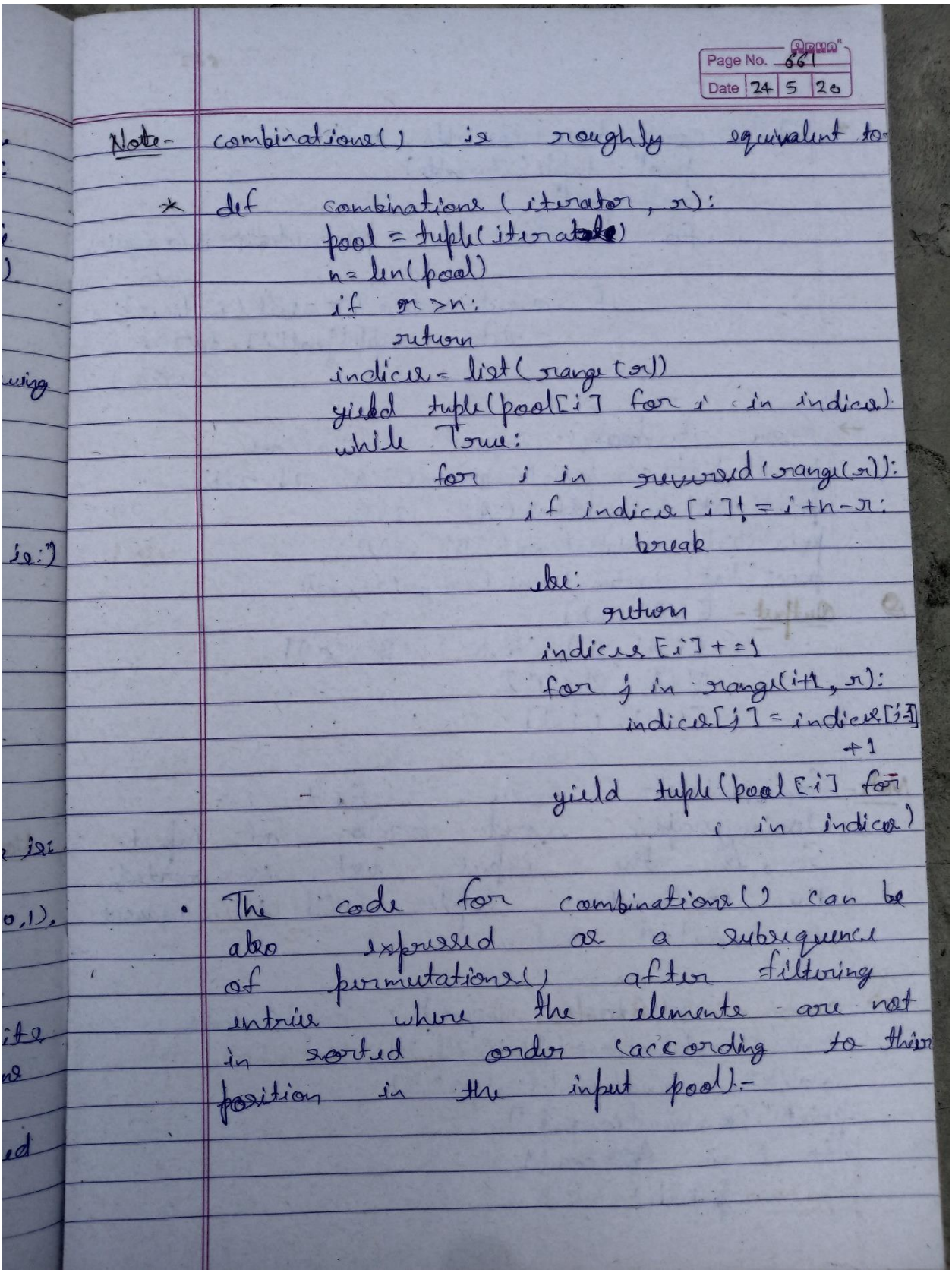
```
        for indices in product(range(n),
                               repeat=r):
            if len(set(indices)) == r:
                yield tuple(pool[i] for i
                            in indices)
```

→ from itertools import permutations
print('Computing all permutation of the following list')
print(list(permutations([3,'Python'],2)))
print('Permutations of following string')
print(list(permutations('AB')))
print('Permutation of the given container is:')
print(list(permutations(range(3),2)))
print(list(permutations(range(3),3)))

Output - Computing all permutations of the following list
[(3,'Python'),('Python',3)]
Permutations of following string
[('A','B'),('B','A')]
Permutation of the given container is:
[(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)]
[(0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1),(2,1,0)]

n.

(iii) combinations(iterable, r) - This iterator prints all possible combinations (without replacement) of the container passed in arguments in the specified group size (r) in sorted order.
Syntax - combinations(iterable, r)

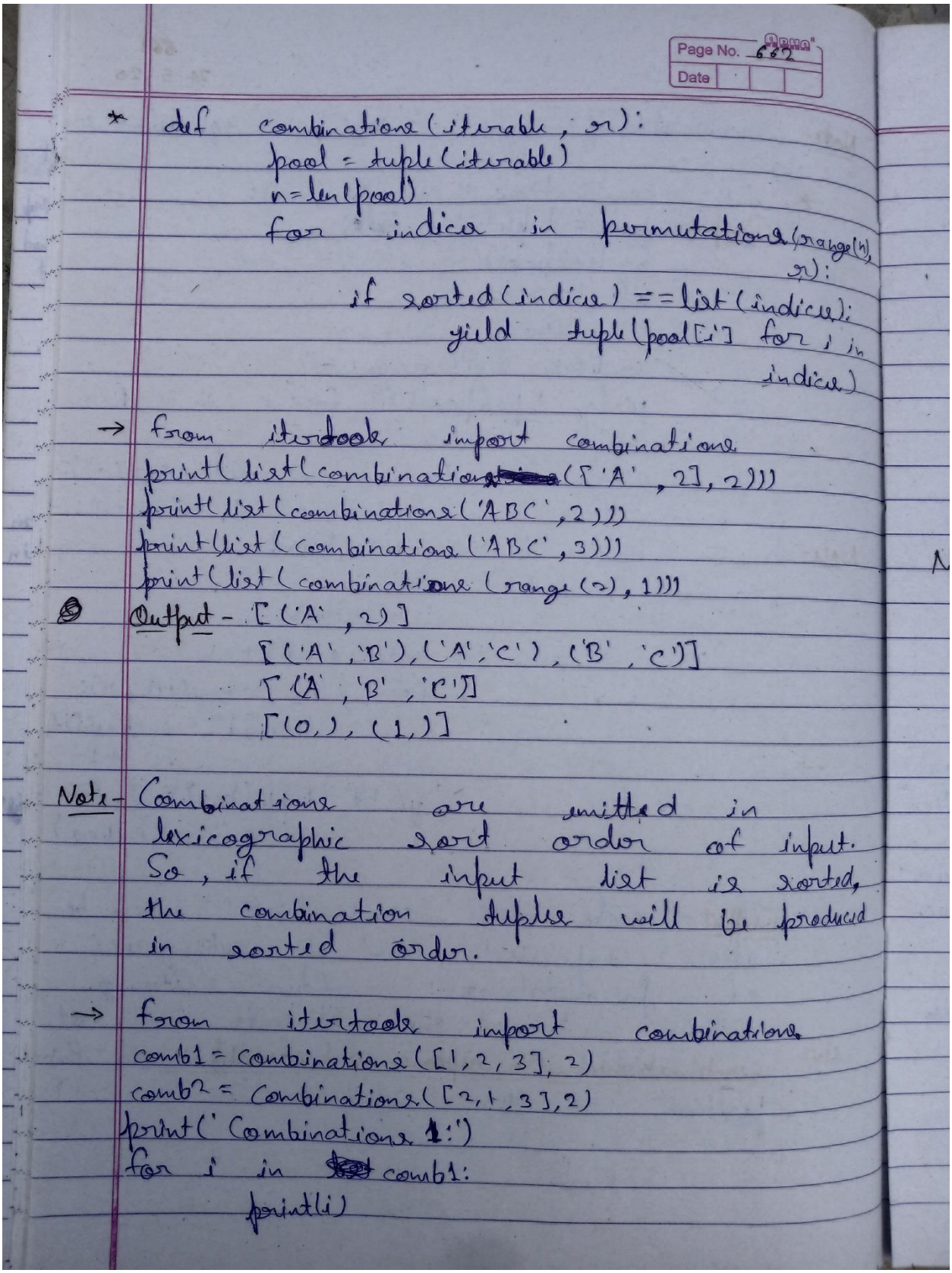


Page No. 66
Date 24 5 20

Note- combinations() is roughly equivalent to:

```
* def combinations(iterator, r):
    pool = tuple(iterator)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

- The code for combinations() can be also expressed as a subsequence of permutations() after filtering entries where the elements are not in sorted order (according to their position in the input pool):-



Page No. 682
Date

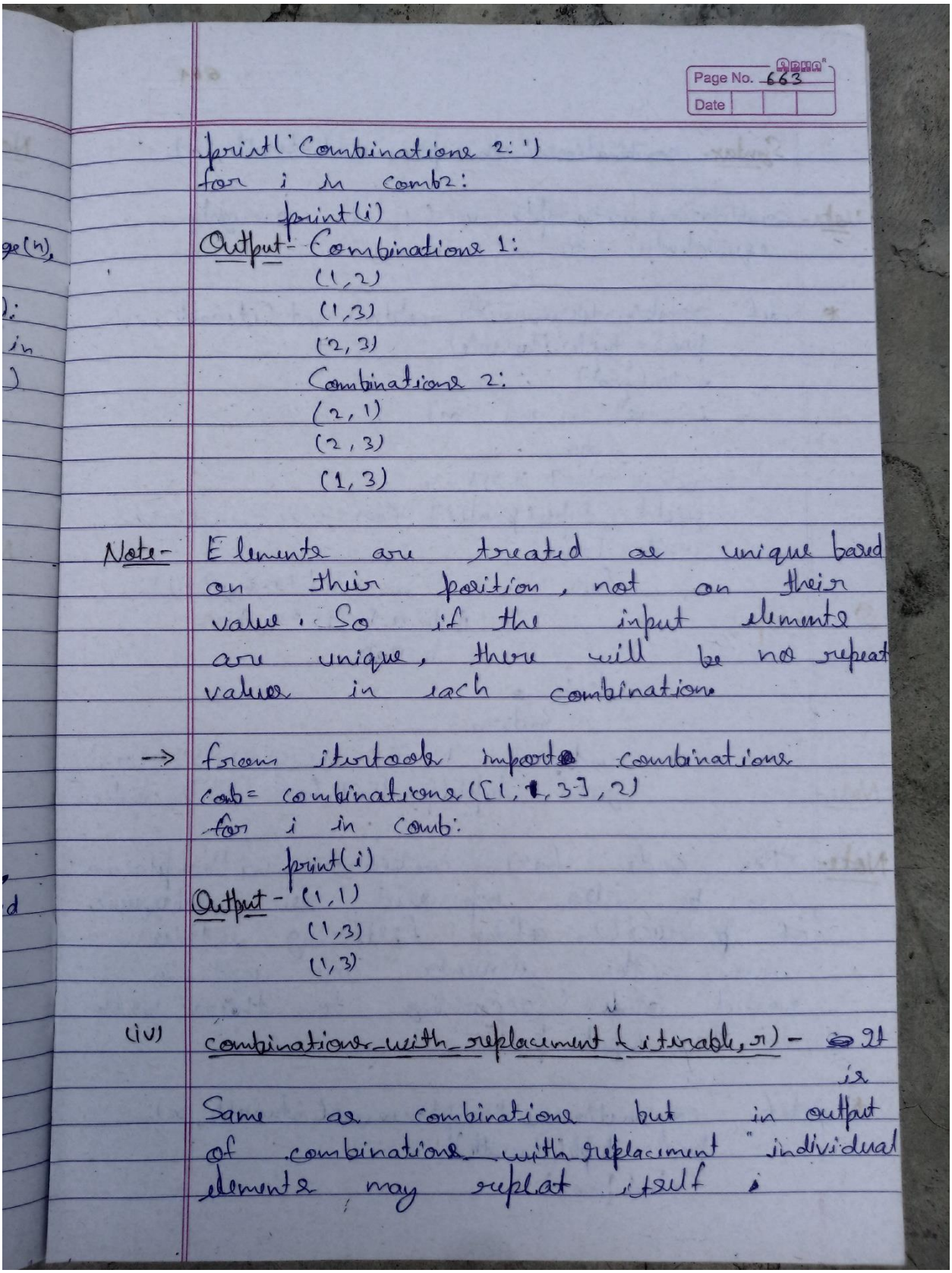
```
* def combinations(iterable, r):
      pool = tuple(iterable)
      n = len(pool)
      for indices in permutations(range(n), r):
          if sorted(indices) == list(indices):
              yield tuple(pool[i] for i in indices)
```

→ from itertools import combinations
print(list(combinations(['A', 2], 2)))
print(list(combinations('ABC', 2)))
print(list(combinations('ABC', 3)))
print(list(combinations(range(2), 1)))

Output - [('A', 2)]
[('A', 'B'), ('A', 'C'), ('B', 'C')]
[('A', 'B', 'C')]
[(0,), (1,)]

Note - Combinations are emitted in lexicographic sort order of input. So, if the input list is sorted, the combination tuples will be produced in sorted order.

→ from itertools import combinations
comb1 = combinations([1, 2, 3], 2)
comb2 = combinations([2, 1, 3], 2)
print('Combinations 1:')
for i in comb1:
    print(i)



Page No. 663
Date

```
print('Combinations 2:')
for i in comb2:
    print(i)
```

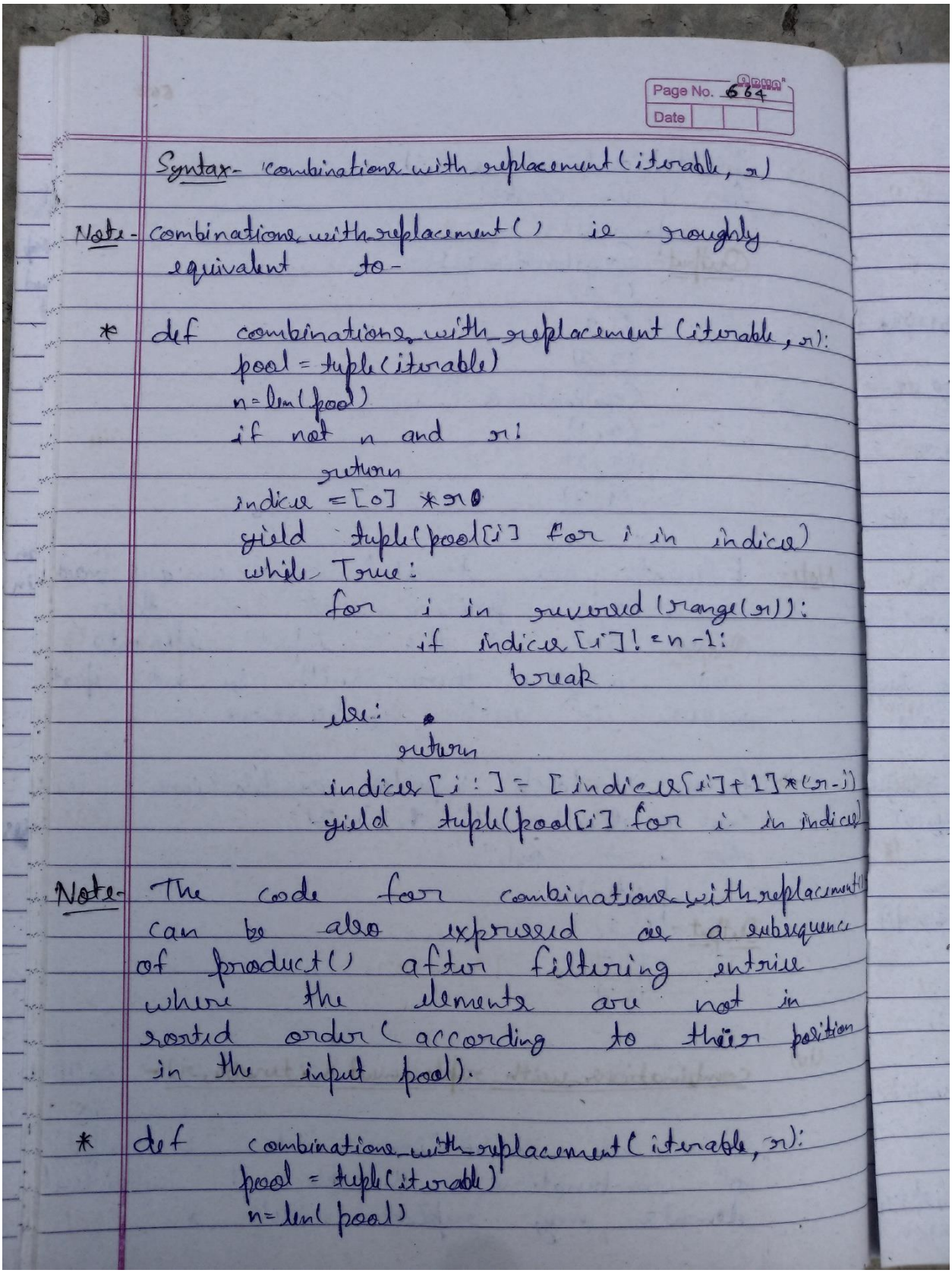
Output- Combinations 1:
(1,2)
(1,3)
(2,3)
Combinations 2:
(2,1)
(2,3)
(1,3)

Note- Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

→
```
from itertools import combinations
comb = combinations([1,1,3], 2)
for i in comb:
    print(i)
```

Output- (1,1)
(1,3)
(1,3)

(iv) combinations_with_replacement(iterable, r) - It is Same as combinations but in output of combinations_with_replacement individual elements may replat itself.



Page No. 564
Date

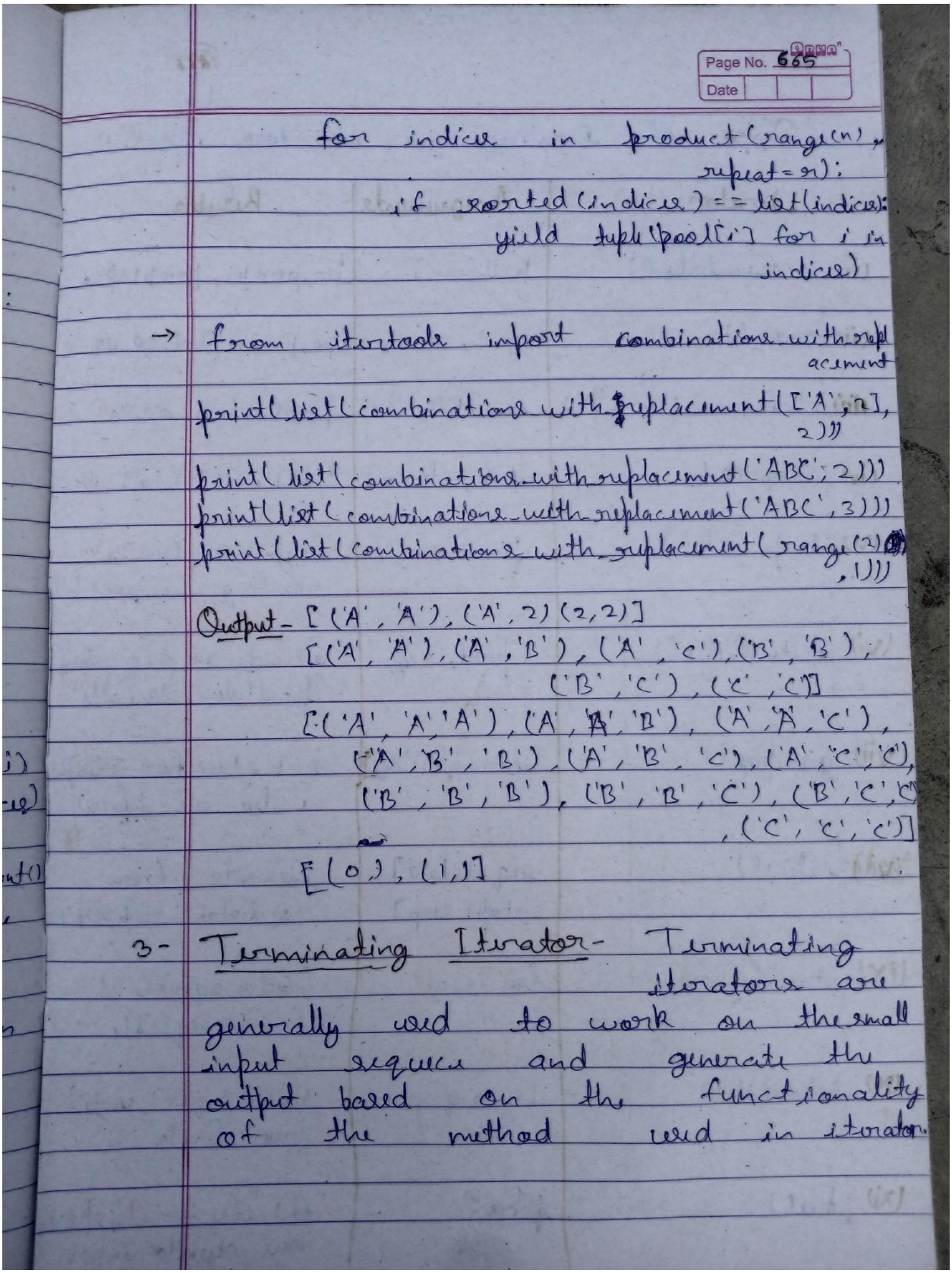
Syntax- combinations_with_replacement(iterable, r)

Note- combinations_with_replacement() is roughly equivalent to-

```
* def combinations_with_replacement(iterable, r):
      pool = tuple(iterable)
      n = len(pool)
      if not n and r:
          return
      indices = [0] * r
      yield tuple(pool[i] for i in indices)
      while True:
          for i in reversed(range(r)):
              if indices[i] != n - 1:
                  break
          else:
              return
          indices[i:] = [indices[i] + 1] * (r - i)
          yield tuple(pool[i] for i in indices)
```

Note- The code for combinations_with_replacement() can be also expressed as a subsequence of product() after filtering entries where the elements are not in sorted order (according to their position in the input pool).

```
* def combinations_with_replacement(iterable, r):
      pool = tuple(iterable)
      n = len(pool)
```



```
    for indices in product(range(n),
                           repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in
                        indices)
```

→ from itertools import combinations_with_replacement

```
print(list(combinations_with_replacement(['A',2],
                                         2)))
print(list(combinations_with_replacement('ABC',2)))
print(list(combinations_with_replacement('ABC',3)))
print(list(combinations_with_replacement(range(3)
                                         ,1)))
```

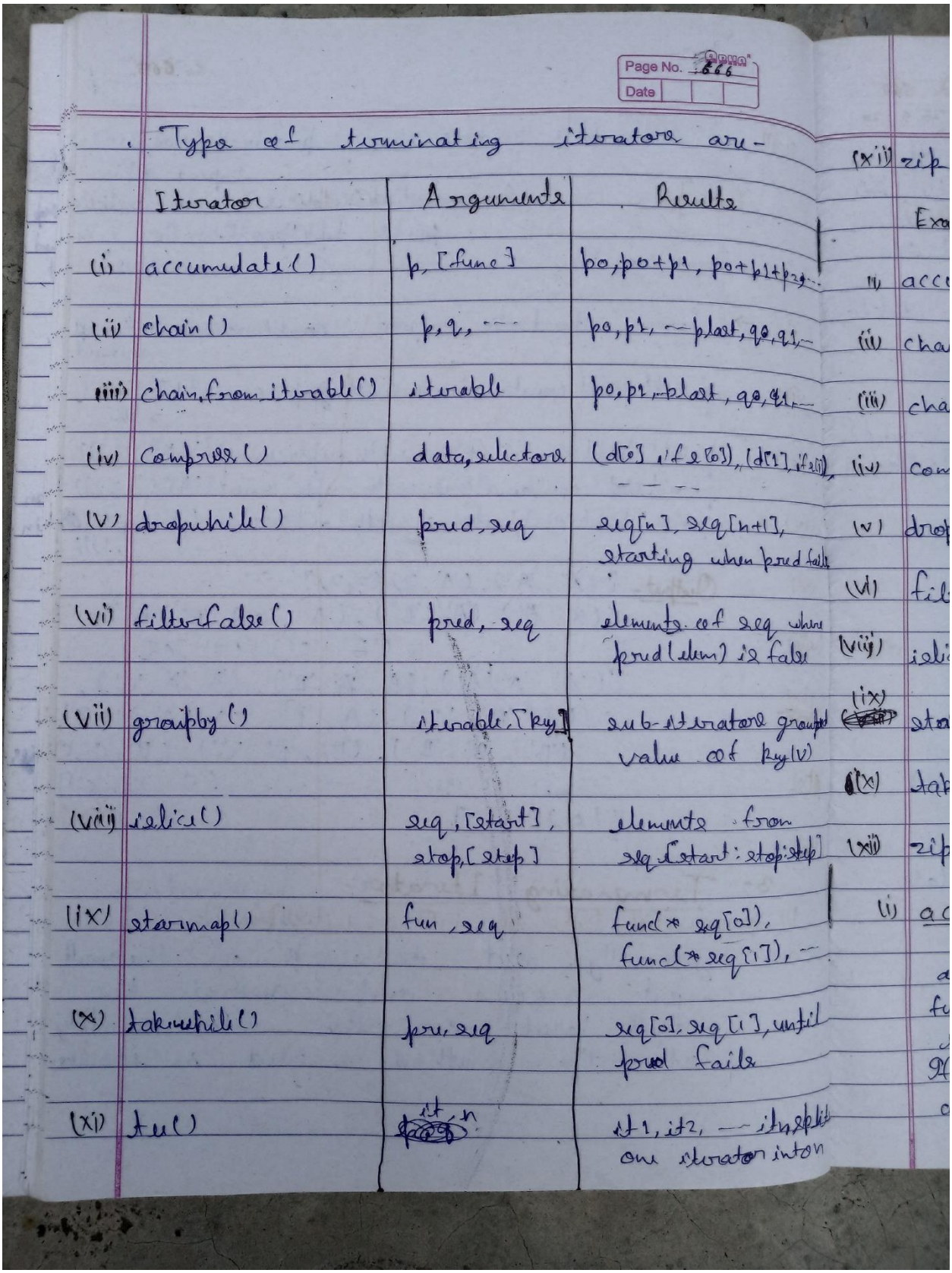
Output- [('A','A'),('A',2),(2,2)]

[('A','A'),('A','B'),('A','C'),('B','B'),
('B','C'),('C','C')]

[('A','A','A'),('A','A','B'),('A','A','C'),
('A','B','B'),('A','B','C'),('A','C','C'),
('B','B','B'),('B','B','C'),('B','C','C'),
('C','C','C')]

[(0,),(1,)]

3- Terminating Iterator- Terminating iterators are generally used to work on the small input sequence and generate the output based on the functionality of the method used in iterator.

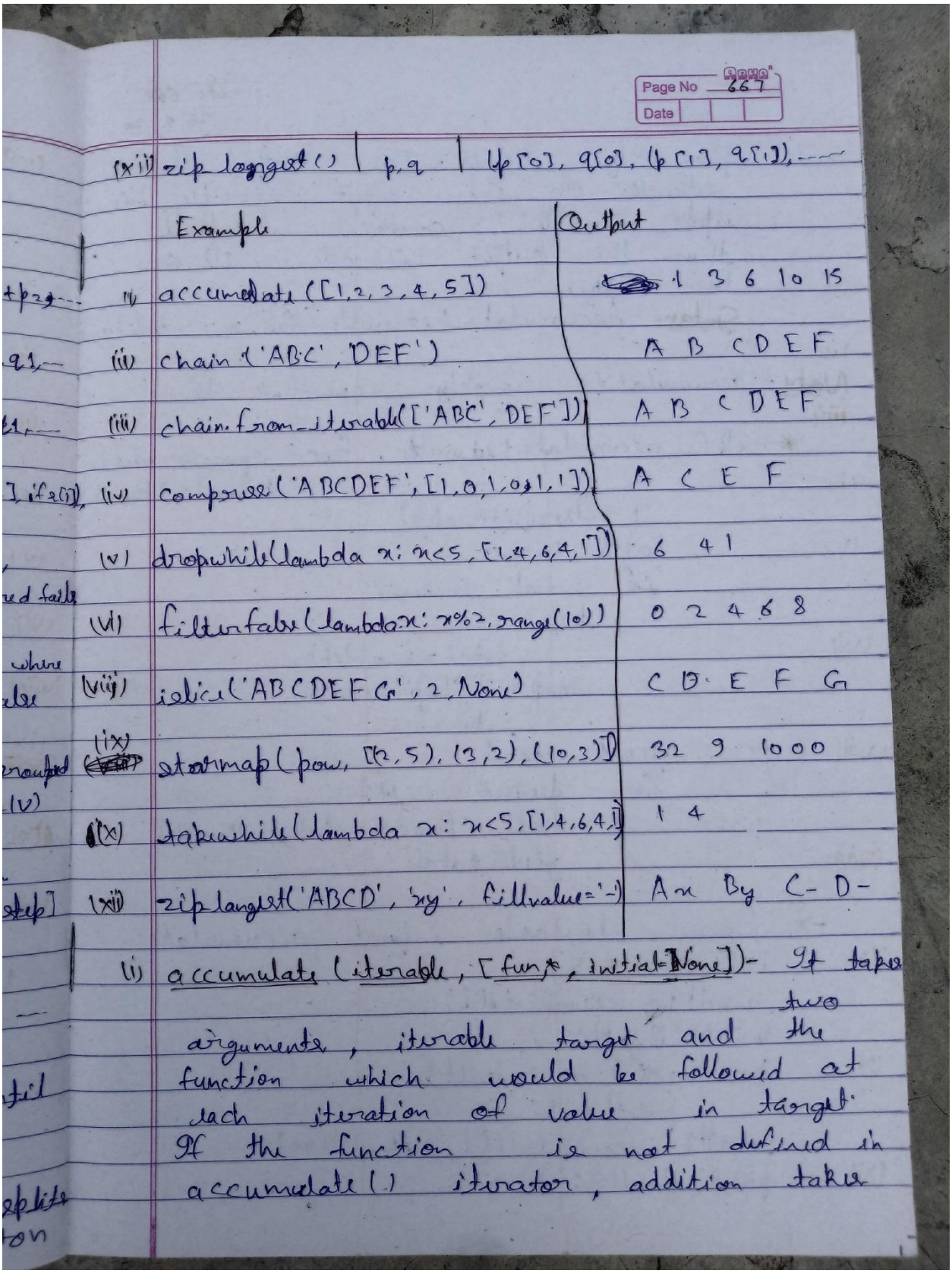


Page No. 866
Date

. Types of terminating iterators are -

	Iterator	Arguments	Results
(i)	accumulate()	p, [func]	p0, p0+p1, p0+p1+p2, ...
(ii)	chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...
(iii)	chain.from_iterable()	iterable	p0, p1, ... plast, q0, q1, ...
(iv)	compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...
(v)	dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails
(vi)	filterfalse()	pred, seq	elements of seq where pred(elem) is false
(vii)	groupby()	iterable, [key]	sub-iterators grouped by value of key(v)
(viii)	islice()	seq, [start], stop, [step]	elements from seq[start:stop:step]
(ix)	starmap()	fun, seq	fun(*seq[0]), fun(*seq[1]), ...
(x)	takewhile()	pre, seq	seq[0], seq[1], until pred fails
(xi)	tee()	it, n	it1, it2, ... itn split one iterator into n

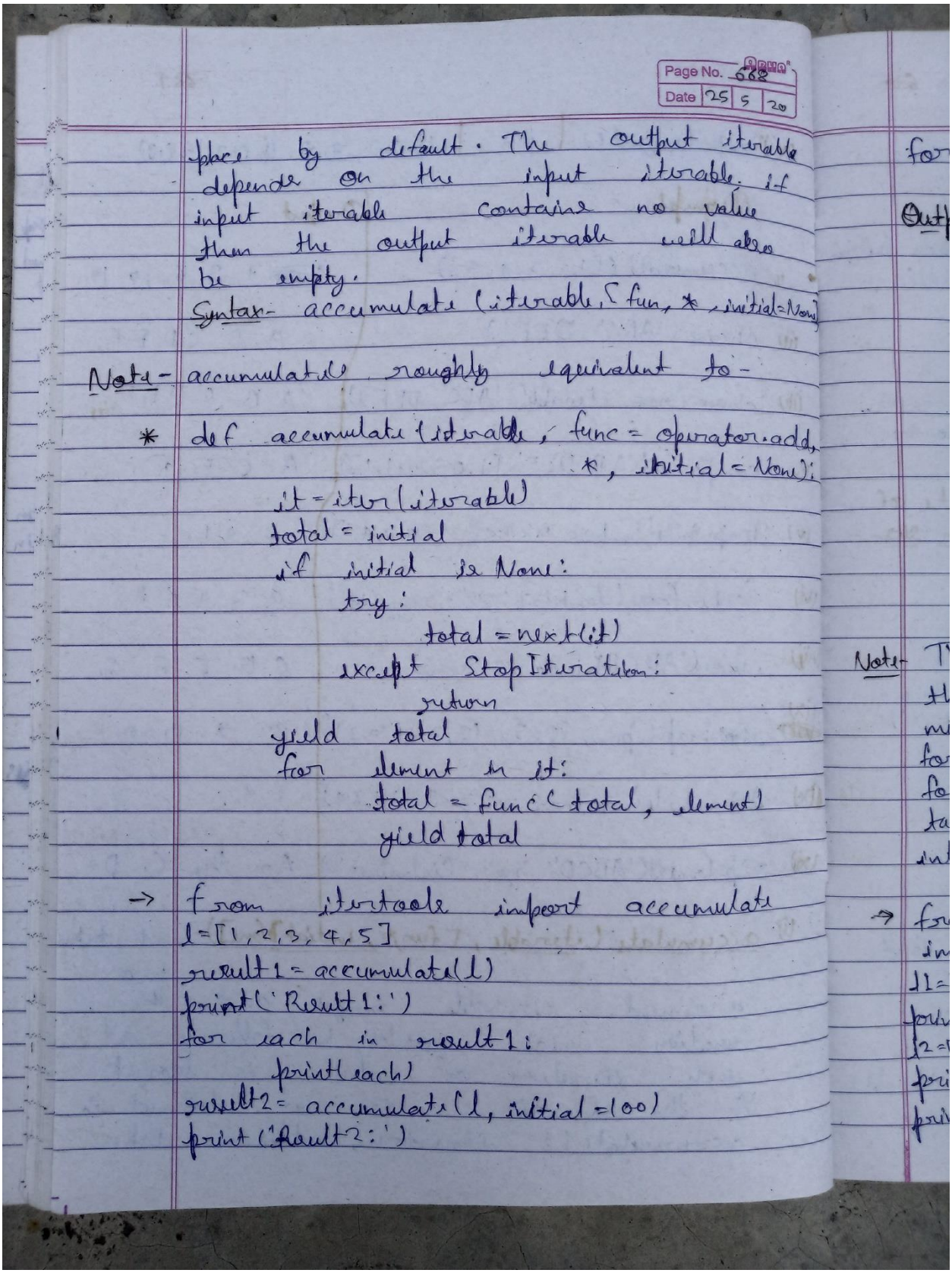
(xii) zip



(xii) zip_longest() | p, q | (p[0], q[0]), (p[1], q[1]), ...

	Example	Output
(i)	accumulate([1,2,3,4,5])	1 3 6 10 15
(ii)	chain('ABC', 'DEF')	A B C D E F
(iii)	chain.from_iterable(['ABC', 'DEF'])	A B C D E F
(iv)	compress('ABCDEF', [1,0,1,0,1,1])	A C E F
(v)	dropwhile(lambda x: x<5, [1,4,6,4,1])	6 4 1
(vi)	filterfalse(lambda x: x%2, range(10))	0 2 4 6 8
(vii)	islice('ABCDEFG', 2, None)	C D E F G
(ix)	starmap(pow, [(2,5), (3,2), (10,3)])	32 9 1000
(x)	takewhile(lambda x: x<5, [1,4,6,4,1])	1 4
(xii)	zip_longest('ABCD', 'xy', fillvalue='-')	Ax By C- D-

(i) accumulate(iterable, [fun*, initial=None]) - It takes two arguments, iterable target and the function which would be followed at each iteration of value in target. If the function is not defined in accumulate() iterator, addition takes



Page No. 552
Date 25 5 20

place by default. The output iterable depends on the input iterable. if input iterable contains no value then the output iterable will also be empty.

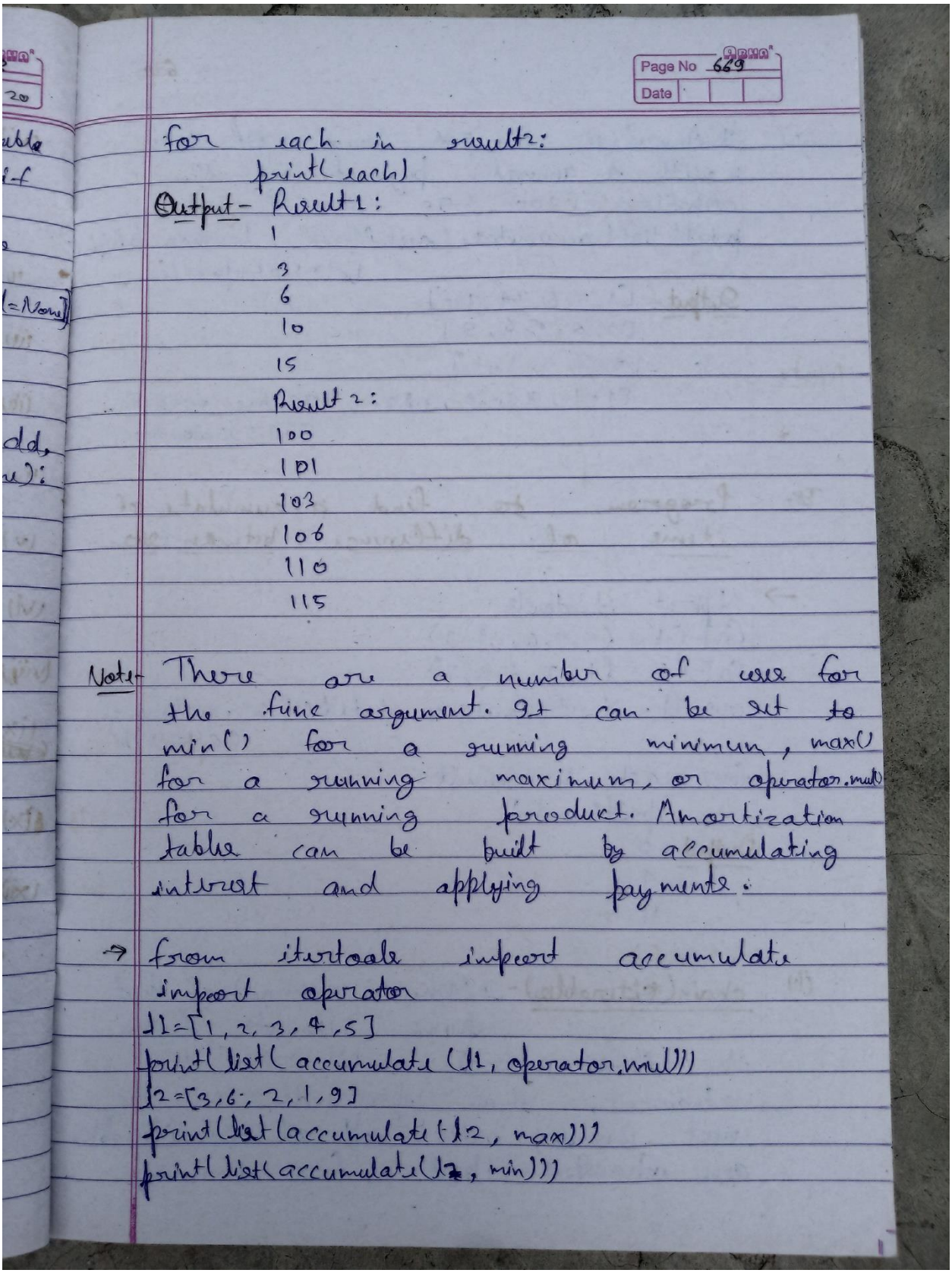
Syntax- accumulate (iterable, [fun, *, initial=None])

Note- accumulate() roughly equivalent to -

```
* def accumulate(iterable, func = operator.add,
                 *, initial = None):
      it = iter(iterable)
      total = initial
      if initial is None:
          try:
              total = next(it)
          except StopIteration:
              return
      yield total
      for element in it:
          total = func(total, element)
          yield total
```

→

```
from itertools import accumulate
l = [1, 2, 3, 4, 5]
result1 = accumulate(l)
print('Result 1:')
for each in result1:
    print(each)
result2 = accumulate(l, initial = 100)
print('Result 2:')
```



```
for each in result2:
    print(each)
```

Output- Result 1:

1
3
6
10
15

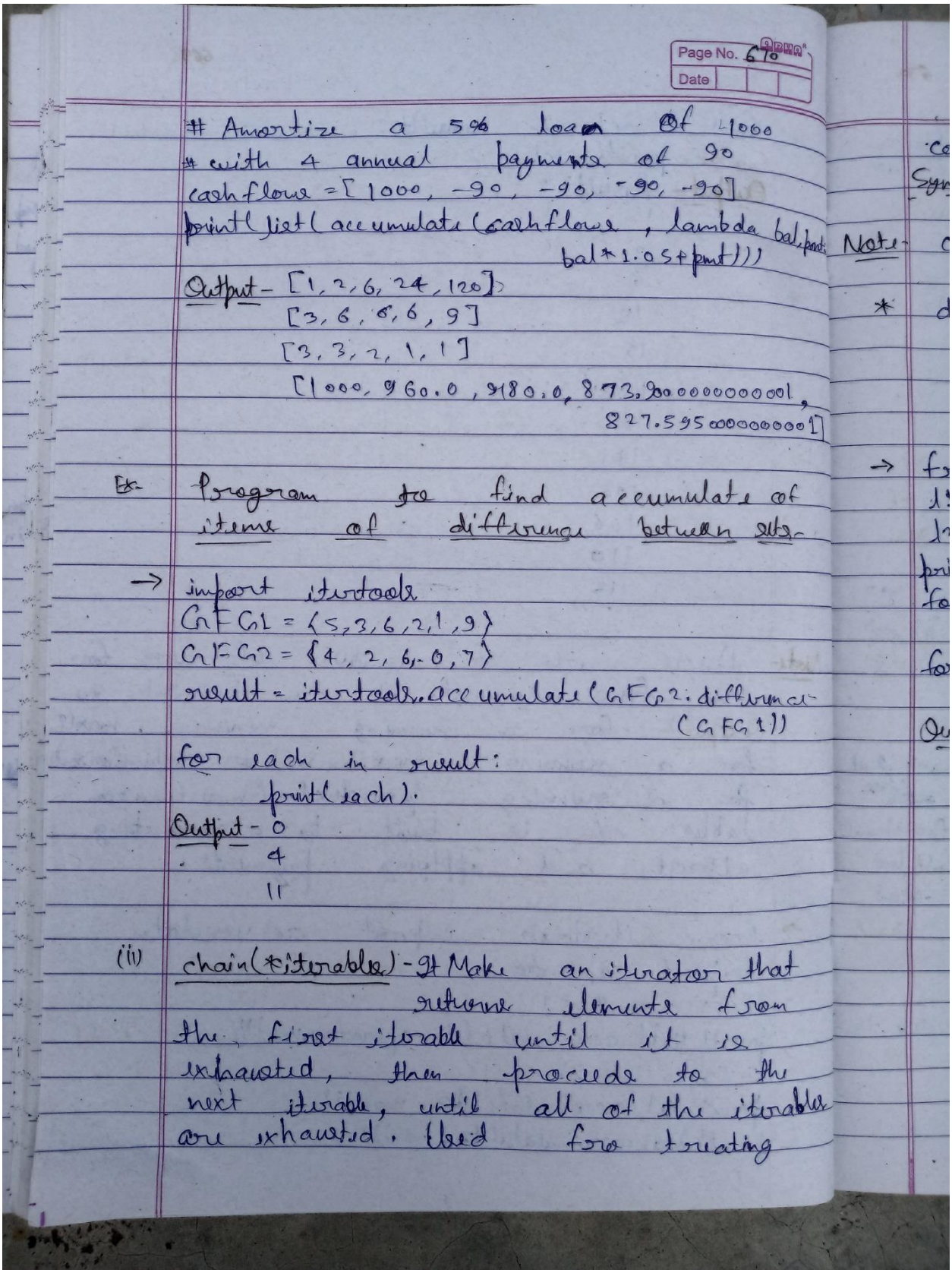
Result 2:

100
101
103
106
110
115

Note- There are a number of uses for the func argument. It can be set to min() for a running minimum, max() for a running maximum, or operator.mul() for a running product. Amortization tables can be built by accumulating interest and applying payments.

→

```
from itertools import accumulate
import operator
l1=[1,2,3,4,5]
print(list(accumulate(l1, operator.mul)))
l2=[3,6,2,1,9]
print(list(accumulate(l2, max)))
print(list(accumulate(l2, min)))
```



Page No. 67
Date

# Amortize a 5% loan of 1000
# with 4 annual payments of 90
cashflows = [1000, -90, -90, -90, -90]
print(list(accumulate(cashflows, lambda bal,pmt: bal*1.05+pmt)))

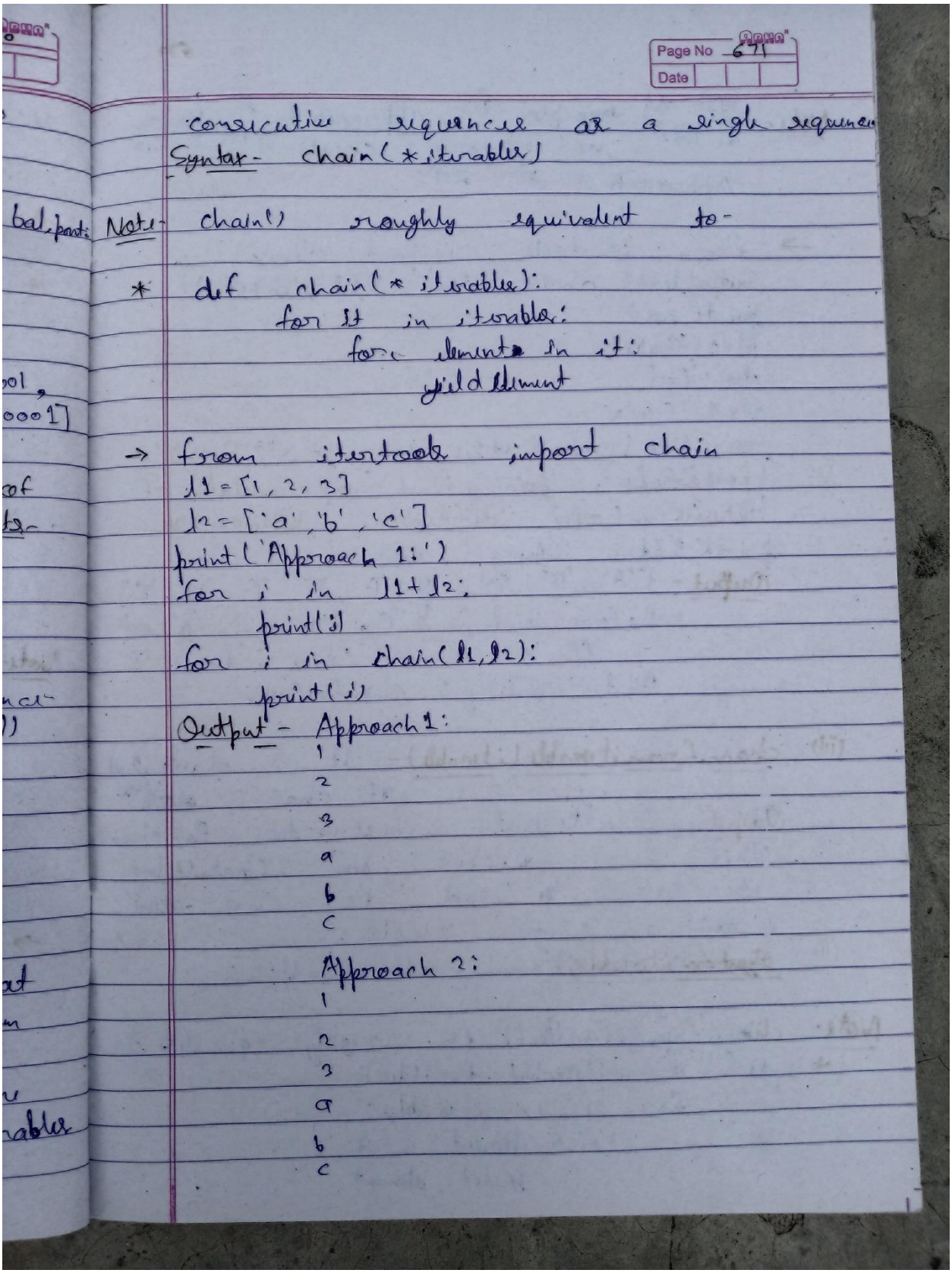
Output - [1, 2, 6, 24, 120]
[3, 6, 8, 6, 9]
[3, 3, 2, 1, 1]
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

Ex Program to find accumulate of items of difference between sets-

→ import itertools
GrFGr1 = {5, 3, 6, 2, 1, 9}
GrFGr2 = {4, 2, 6, 0, 7}
result = itertools.accumulate(GrFGr2 : difference (GrFGr1))
for each in result:
    print(each).

Output - 0
4
11

(ii) chain(*iterables) - It Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating



consecutive sequences as a single sequence

Syntax- chain(*iterables)

Note- chain() roughly equivalent to-

```
* def chain(*iterables):
      for it in iterables:
          for element in it:
              yield element
```

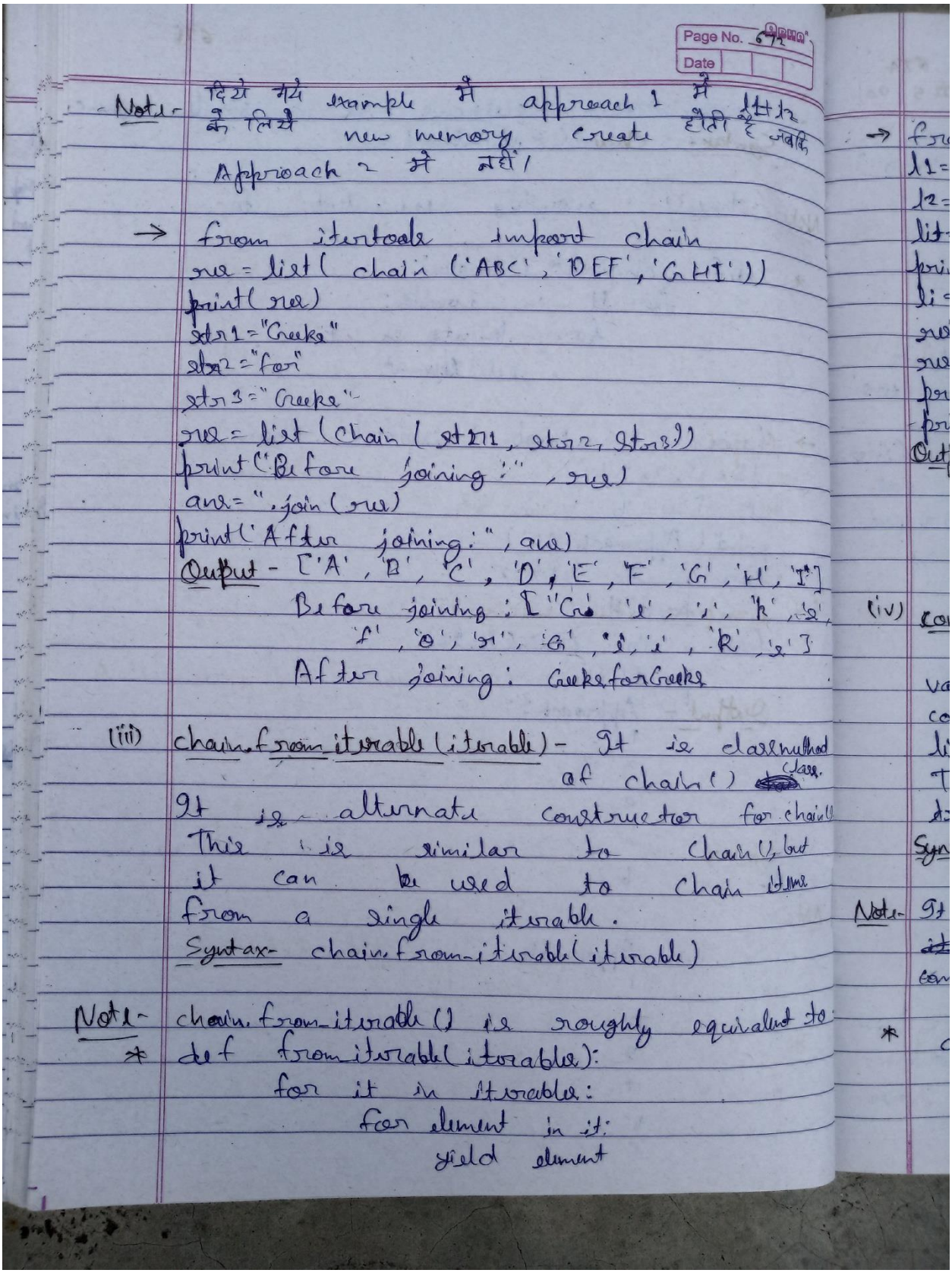
→
```
from itertools import chain
l1 = [1, 2, 3]
l2 = ['a', 'b', 'c']
print('Approach 1:')
for i in l1+l2:
    print(i)
for i in chain(l1, l2):
    print(i)
```

Output - Approach 1:

1
2
3
a
b
c

Approach 2:

1
2
3
a
b
c



Note- दिये गये example में approach 1 में l1+l2 के लिये new memory create होती है जबकि Approach 2 में नहीं।

→ from itertools import chain
res = list(chain ('ABC', 'DEF', 'GHI'))
print(res)
str1 = "Geeks"
str2 = "for"
str3 = "Geeks"
res = list (chain (str1, str2, str3))
print ("Before joining :", res)
ans = ''.join(res)
print('After joining:", ans)
Output - ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
Before joining : ['G', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'G', 'e', 'e', 'k', 's']
After joining: GeeksforGeeks

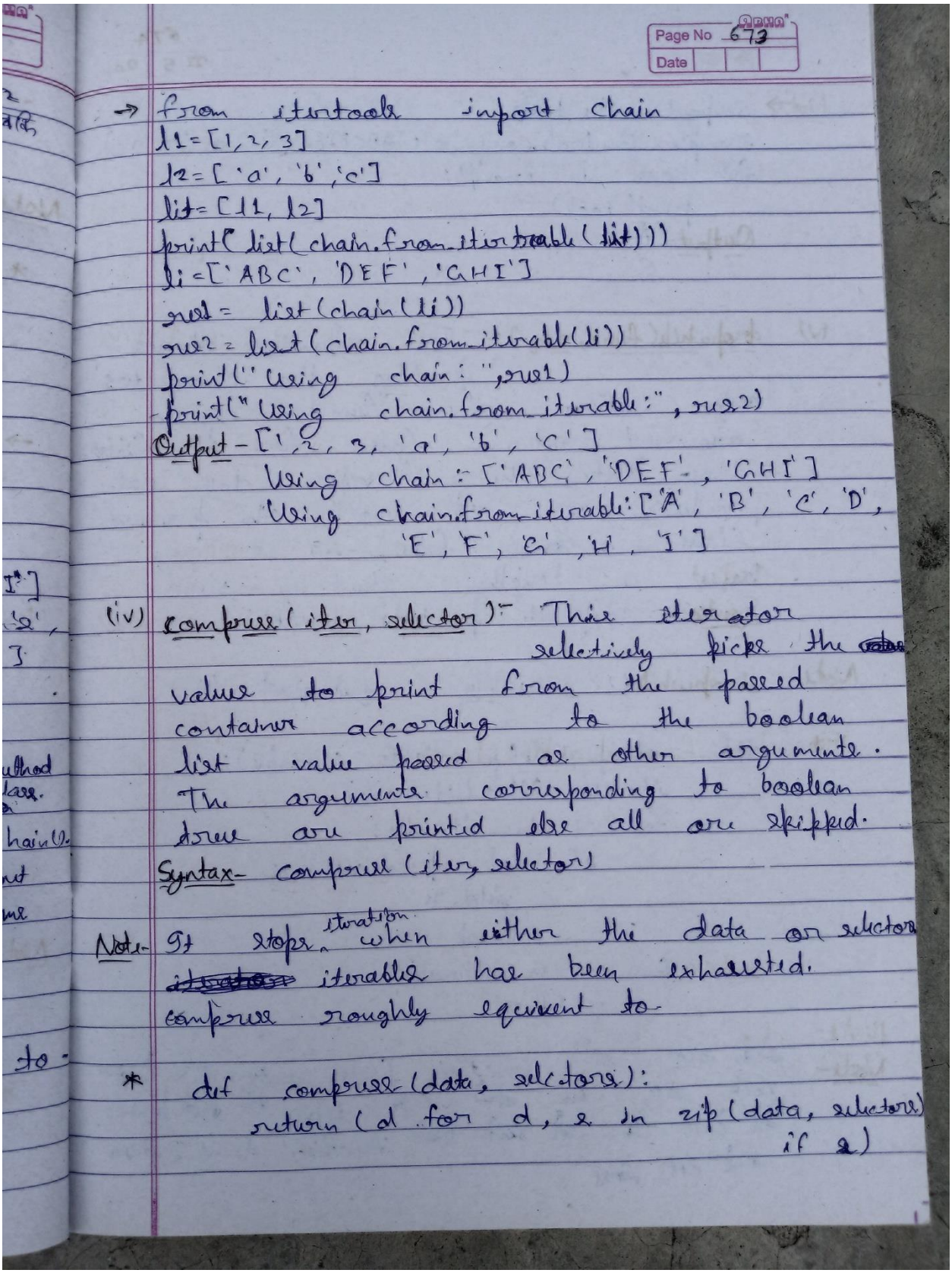
(iii) chain.from_iterable(iterable) - It is classmethod of chain() class.
It is alternate constructor for chain(). This is similar to chain(), but it can be used to chain items from a single iterable.
Syntax- chain.from_iterable(iterable)

Note- chain.from_iterable() is roughly equivalent to
* def from_iterable(iterables):
    for it in iterables:
        for element in it:
            yield element

(iv)

Note- It

*

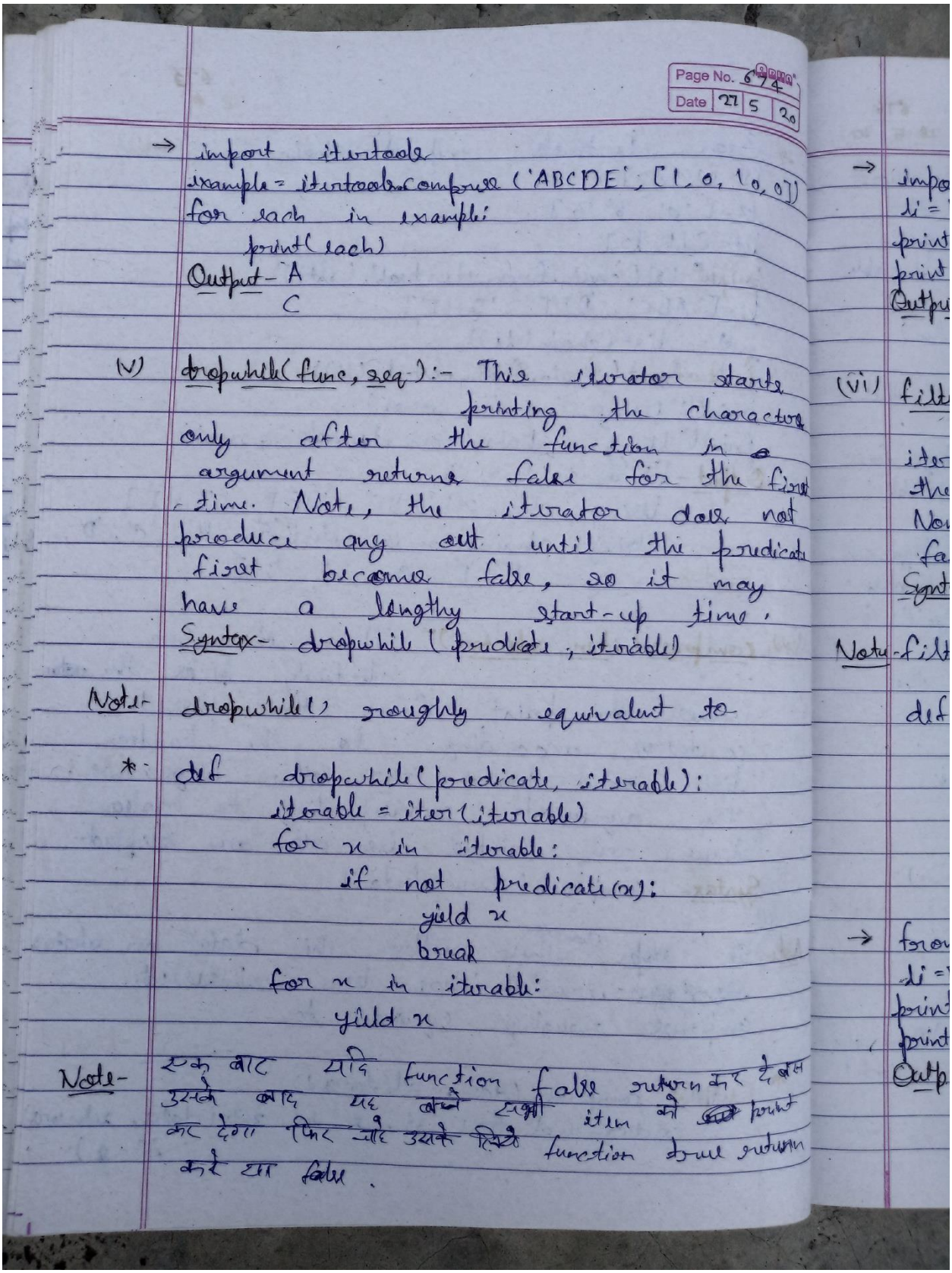


→ from itertools import chain
l1 = [1, 2, 3]
l2 = ['a', 'b', 'c']
lit = [l1, l2]
print(list(chain.from_iterable(lit)))
li = ['ABC', 'DEF', 'GHI']
res1 = list(chain(li))
res2 = list(chain.from_iterable(li))
print("Using chain:", res1)
print("Using chain.from_iterable:", res2)
Output - [1, 2, 3, 'a', 'b', 'c']
Using chain: ['ABC', 'DEF', 'GHI']
Using chain.from_iterable: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']

(iv) compress(iter, selector):- This iterator selectively picks the value to print from the passed container according to the boolean list value passed as other arguments. The arguments corresponding to boolean true are printed else all are skipped.
Syntax- compress(iter, selector)

Note- It stops iteration when either the data or selectors iterables has been exhausted.
compress roughly equivalent to-

```
* def compress(data, selectors):
      return (d for d, s in zip(data, selectors)
                  if s)
```



→ import itertools
example = itertools.compress('ABCDE', [1, 0, 1, 0, 0])
for each in example:
    print(each)
Output- A
        C

(v) dropwhile(func, seq):- This iterator starts printing the characters only after the function in argument returns false for the first time. Note, the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time.
Syntax- dropwhile(predicate, iterable)

Note- dropwhile() roughly equivalent to

*- def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x

Note- एक बार यदि function false return कर दे तब उसके बाद यह बचे हुई item को print कर देगा फिर चाहे उसके लिये function true return करे या false.

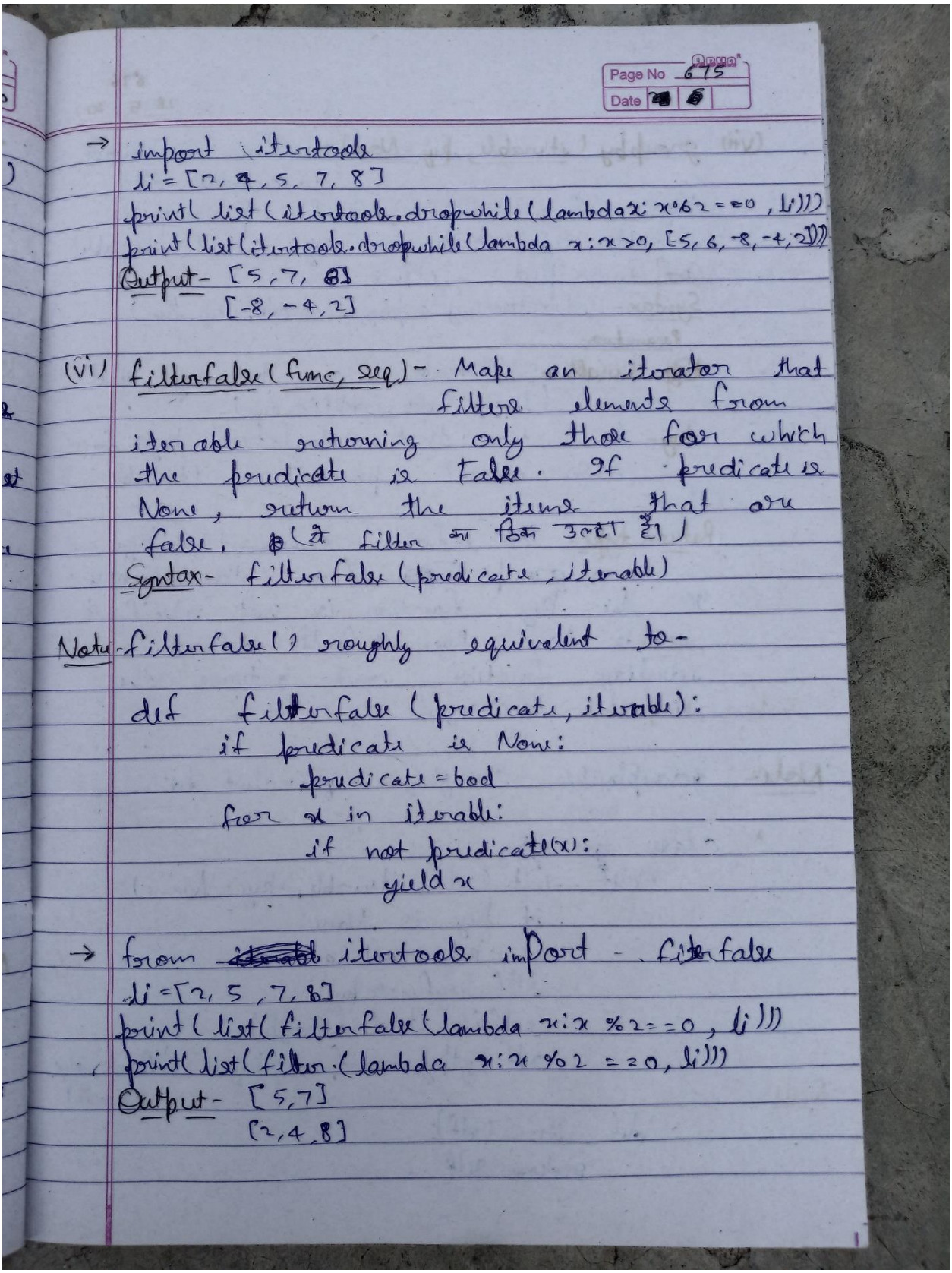
(vi) filt

Note- filt

def

→ fro

Outp



Page No 675
Date 6

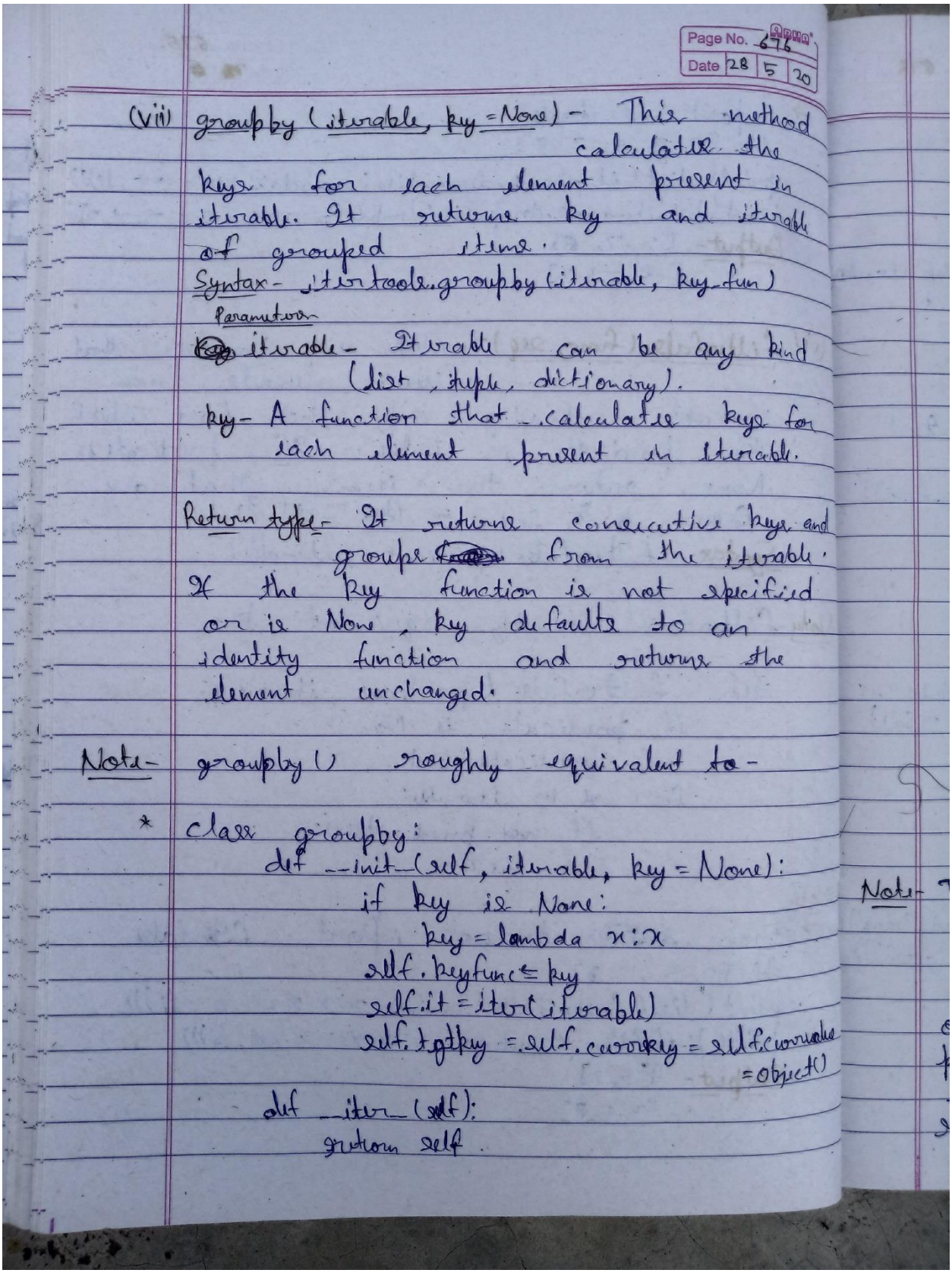
→ import itertools
li = [2, 4, 5, 7, 8]
print(list (itertools.dropwhile (lambda x: x%2 == 0 , li)))
print (list (itertools.dropwhile (lambda x: x > 0, [5, 6, -8, -4, 2])))
Output- [5, 7, 8]
[-8, -4, 2]

(vi) filterfalse (func, seq) - Make an iterator that filters elements from iterable returning only those for which the predicate is False. If predicate is None, return the items that are false. (It filter का ठीक उल्टा है)
Syntax- filterfalse (predicate , iterable)

Note- filterfalse() roughly equivalent to -

```
def filterfalse ( predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

→ from itertools import filterfalse
li = [2, 5, 7, 8]
print (list (filterfalse (lambda x: x % 2 == 0 , li)))
print (list (filter (lambda x: x % 2 == 0, li)))
Output- [5, 7]
[2, 4, 8]



Page No. 676
Date 28 5 20

(vii) groupby (iterable, key=None) - This method calculates the keys for each element present in iterable. It returns key and iterable of grouped items.

Syntax - itertools.groupby (iterable, key_fun)

Parameters

iterable - Iterable can be any kind (list, tuple, dictionary).

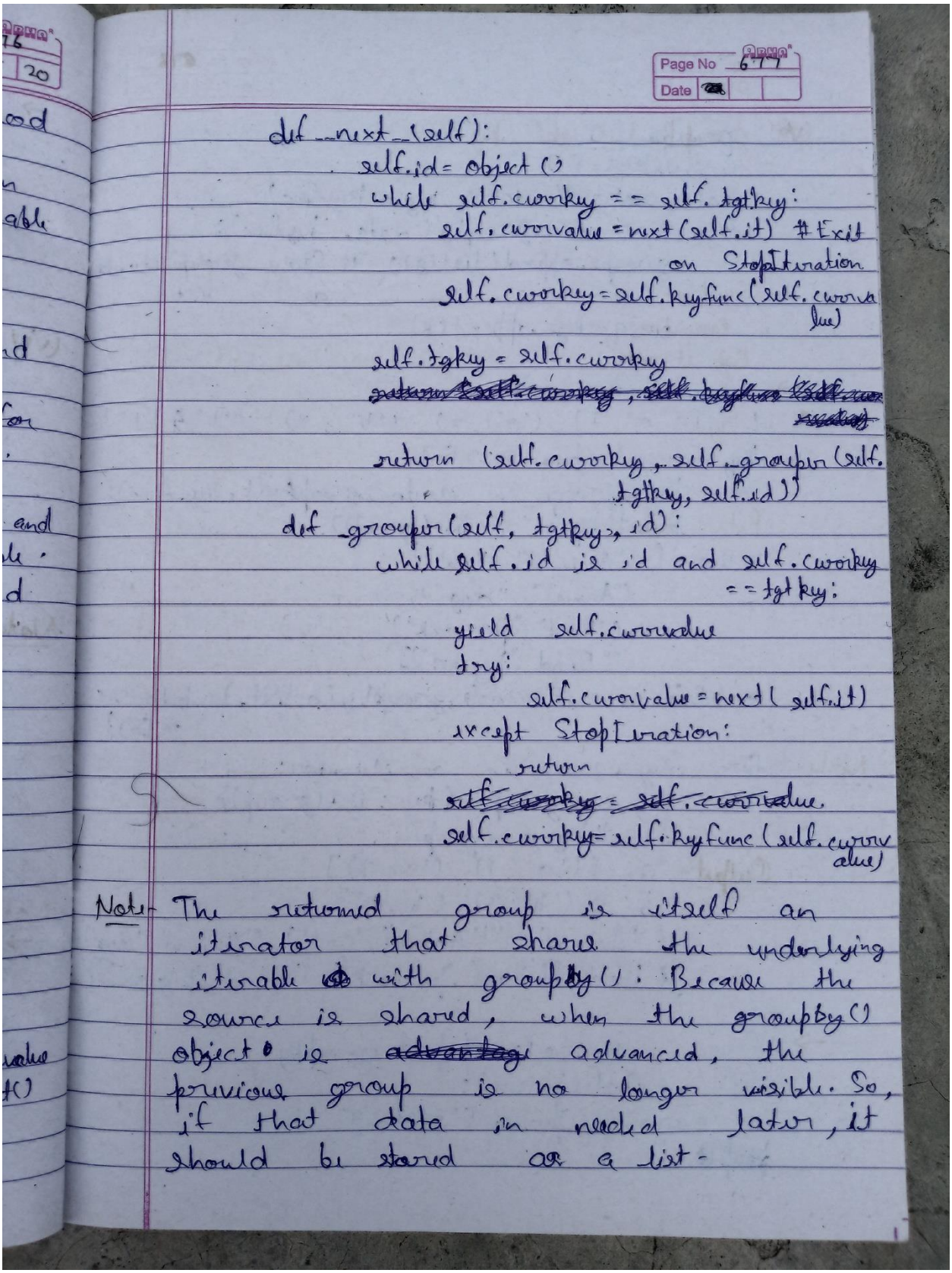
key - A function that calculates keys for each element present in iterable.

Return type - It returns consecutive keys and groups from the iterable. If the key function is not specified or is None, key defaults to an identity function and returns the element unchanged.

Note - groupby() roughly equivalent to -

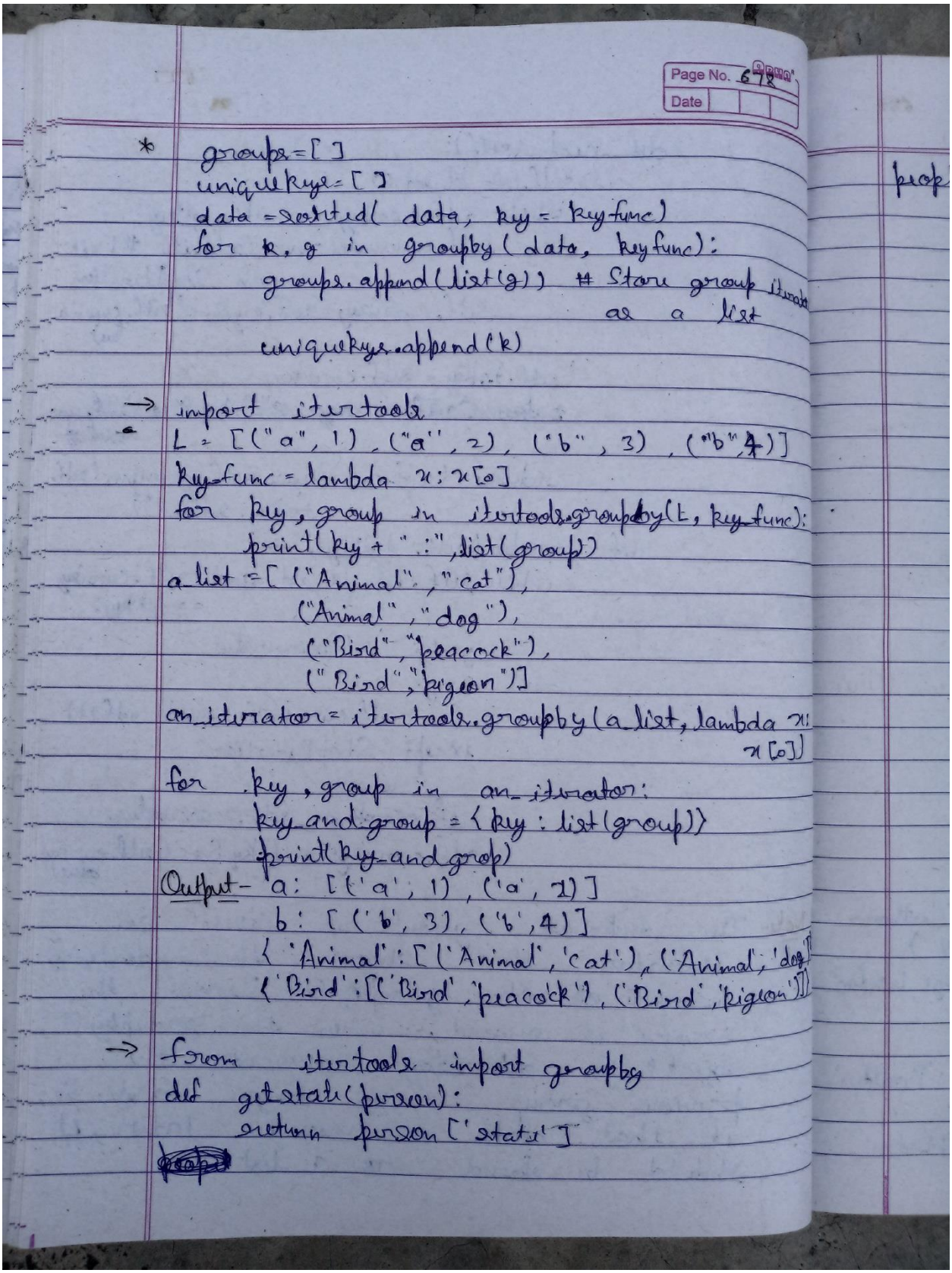
```
* class groupby:
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue
                    = object()

    def __iter__(self):
        return self
```

Note -



```
def __next__(self):
    self.id = object()
    while self.currkey == self.tgtkey:
        self.currvalue = next(self.it) # Exit on StopIteration
        self.currkey = self.keyfunc(self.currvalue)
    self.tgtkey = self.currkey
    return (self.currkey, self._grouper(self.tgtkey, self.id))
def _grouper(self, tgtkey, id):
    while self.id is id and self.currkey == tgtkey:
        yield self.currvalue
        try:
            self.currvalue = next(self.it)
        except StopIteration:
            return
        self.currkey = self.keyfunc(self.currvalue)
```

Note- The returned group is itself an iterator that shares the underlying iterable with groupby(). Because the source is shared, when the groupby() object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list.



```
groups = []
uniquekys = []
data = sorted(data, key = keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))  # Store group iterator as a list
    uniquekys.append(k)
```

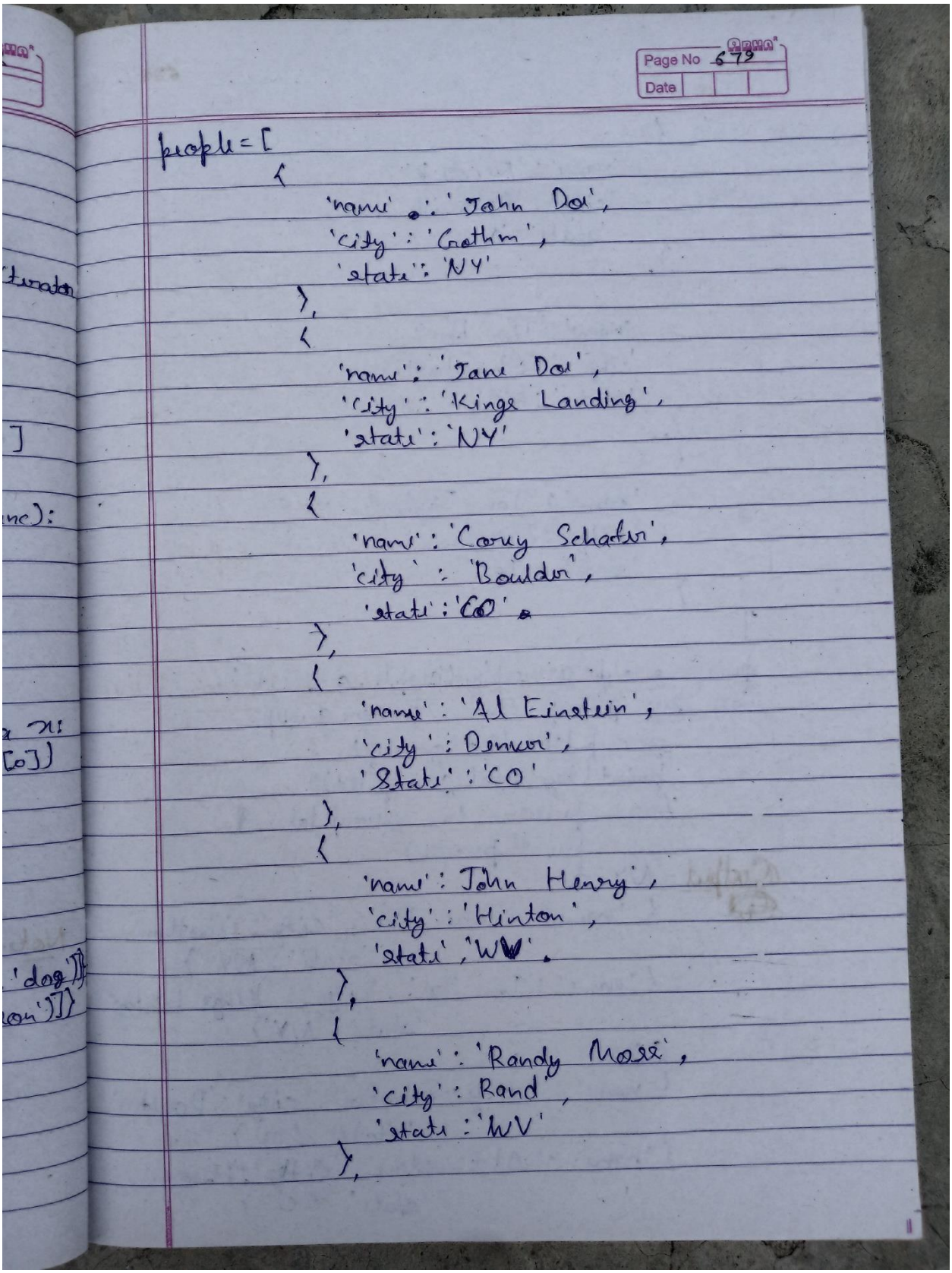
→ import itertools

```
L = [("a", 1), ("a", 2), ("b", 3), ("b", 4)]
key_func = lambda x: x[0]
for key, group in itertools.groupby(L, key_func):
    print(key + " :", list(group))
a_list = [("Animal", "cat"),
          ("Animal", "dog"),
          ("Bird", "peacock"),
          ("Bird", "pigeon")]
an_iterator = itertools.groupby(a_list, lambda x:
                                x[0])
for key, group in an_iterator:
    key_and_group = {key : list(group)}
    print(key_and_group)
```

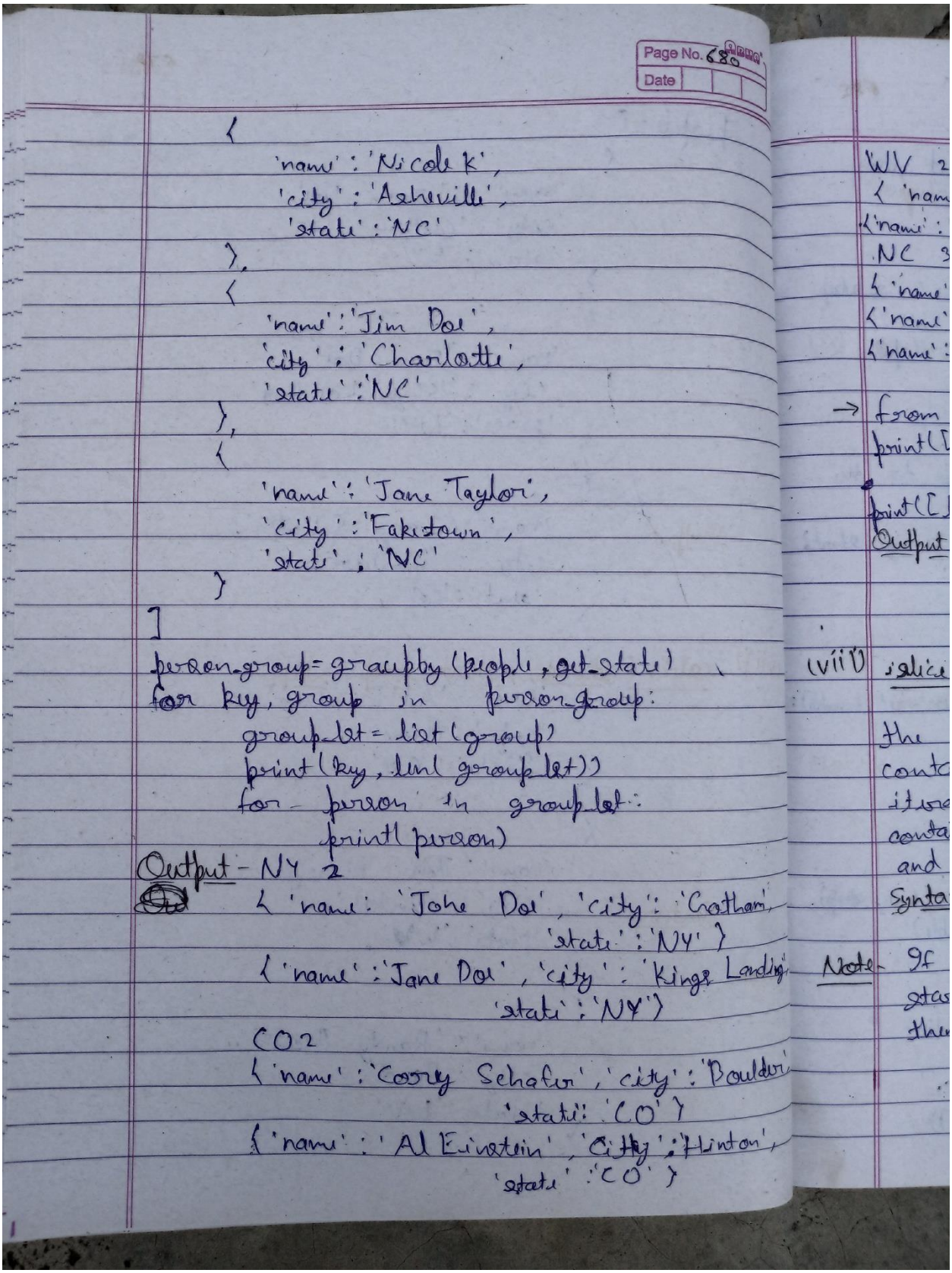
Output- a: [('a', 1), ('a', 2)]
b: [('b', 3), ('b', 4)]
{'Animal': [('Animal', 'cat'), ('Animal', 'dog')]}
{'Bird': [('Bird', 'peacock'), ('Bird', 'pigeon')]}

→

```
from itertools import groupby
def get_state(person):
    return person['state']
```



```
people = [
    {
        'name': 'John Doe',
        'city': 'Gotham',
        'state': 'NY'
    },
    {
        'name': 'Jane Doe',
        'city': 'Kings Landing',
        'state': 'NY'
    },
    {
        'name': 'Corey Schafer',
        'city': 'Boulder',
        'state': 'CO'
    },
    {
        'name': 'Al Einstein',
        'city': 'Denver',
        'State': 'CO'
    },
    {
        'name': 'John Henry',
        'city': 'Hinton',
        'state': 'WV'
    },
    {
        'name': 'Randy Moss',
        'city': 'Rand',
        'state': 'WV'
    },
```



```
{
    'name': 'Nicole K',
    'city': 'Asheville',
    'state': 'NC'
},
{
    'name': 'Jim Doe',
    'city': 'Charlotte',
    'state': 'NC'
},
{
    'name': 'Jane Taylor',
    'city': 'Faketown',
    'state': 'NC'
}
]

person_group = groupby(people, get_state)
for key, group in person_group:
    group_lst = list(group)
    print(key, len(group_lst))
    for person in group_lst:
        print(person)
```

Output - NY 2

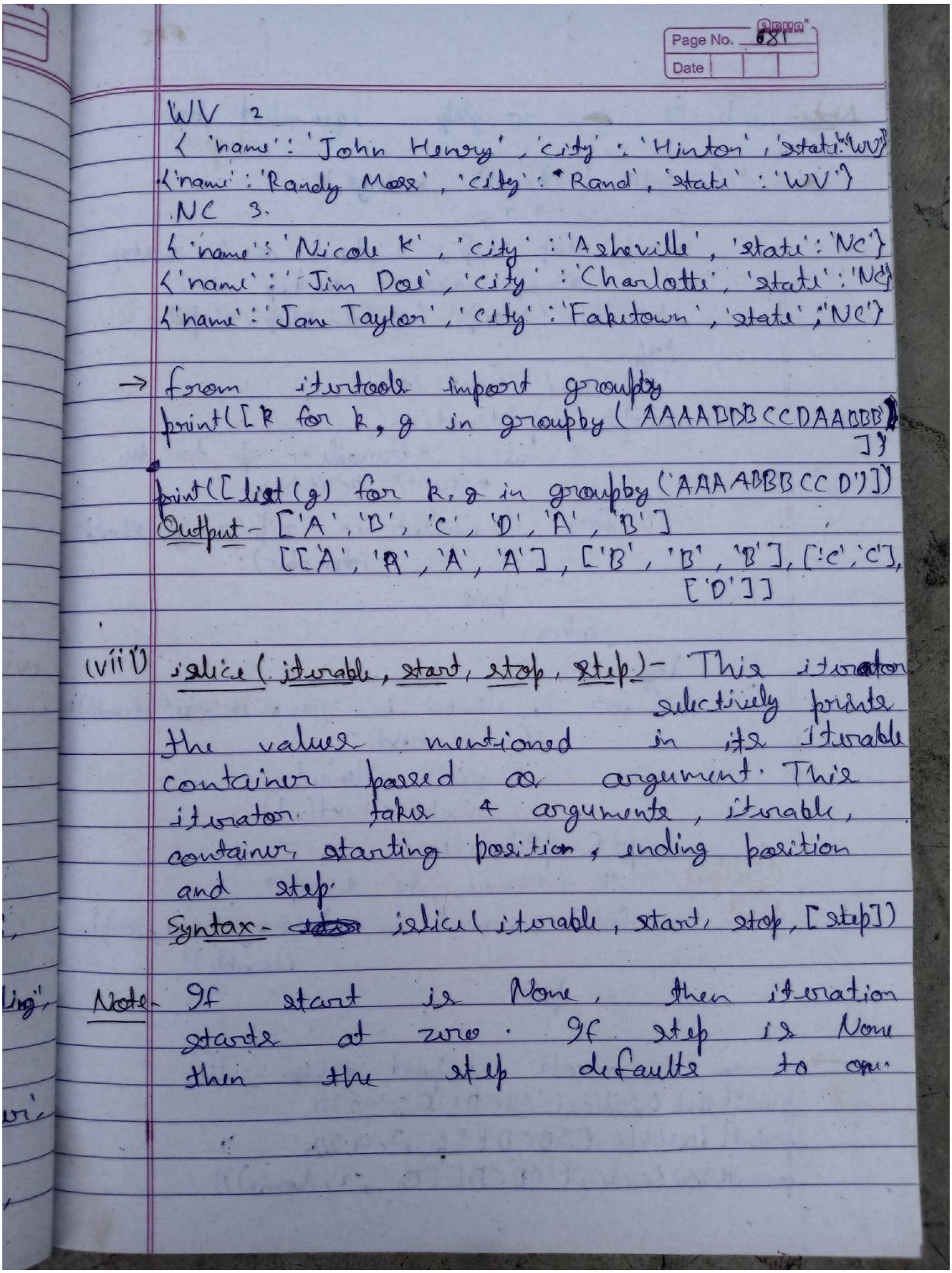
{'name': 'John Doe', 'city': 'Gotham', 'state': 'NY'}

{'name': 'Jane Doe', 'city': 'Kings Landing', 'state': 'NY'}

CO 2

{'name': 'Corey Schafer', 'city': 'Boulder', 'state': 'CO'}

{'name': 'Al Einstein', 'city': 'Hinton', 'state': 'CO'}



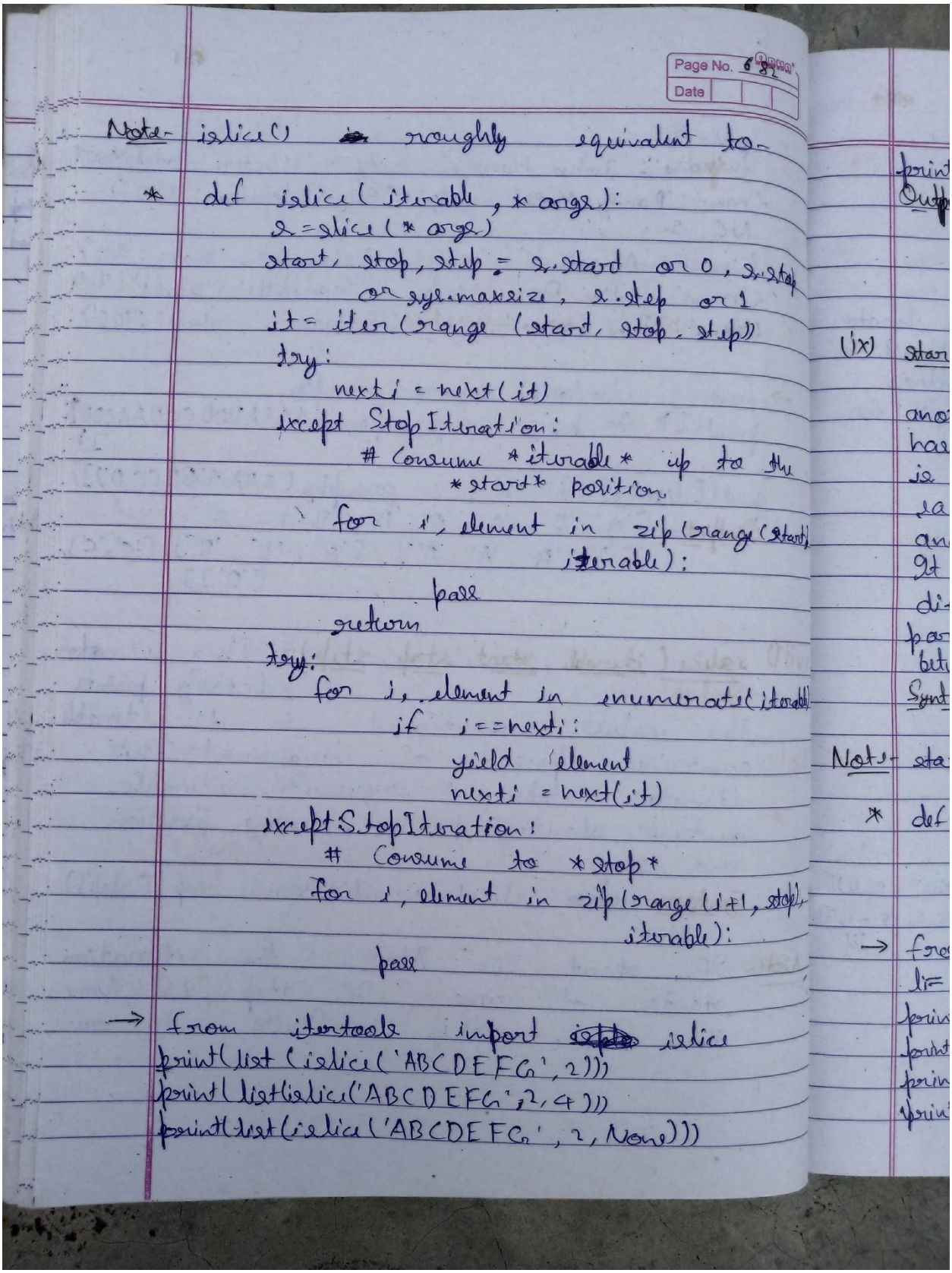
Page No. 681
Date

WV 2
{'name': 'John Henry', 'city': 'Hinton', 'state': 'WV'}
{'name': 'Randy Moss', 'city': 'Rand', 'state': 'WV'}
NC 3.
{'name': 'Nicole K', 'city': 'Asheville', 'state': 'NC'}
{'name': 'Jim Doe', 'city': 'Charlotte', 'state': 'NC'}
{'name': 'Jane Taylor', 'city': 'Faketown', 'state': 'NC'}

→ from itertools import groupby
print([k for k, g in groupby('AAAABBBCCDAABBB')])
print([list(g) for k, g in groupby('AAAABBBCCD')])
Output - ['A', 'B', 'C', 'D', 'A', 'B']
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D']]

(viii) islice(iterable, start, stop, step) - This iterator selectively prints the values mentioned in its iterable container passed as argument. This iterator takes 4 arguments, iterable, container, starting position, ending position and step.
Syntax - islice(iterable, start, stop, [step])

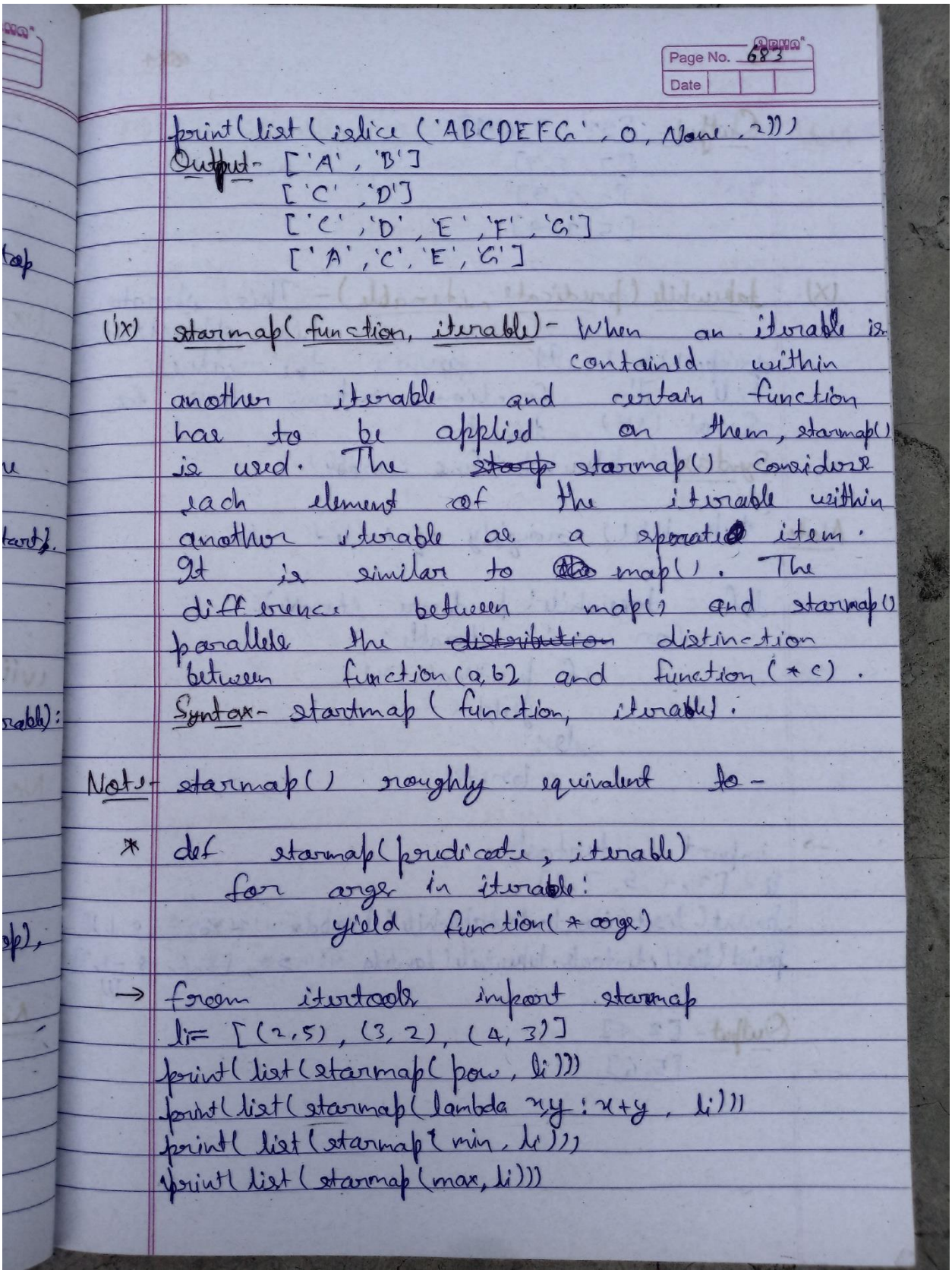
Note - If start is None, then iteration starts at zero. If step is None then the step defaults to one.



Note- islice() is roughly equivalent to-

```
* def islice(iterable, *args):
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop
        or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the
            *start* position.
        for i, element in zip(range(start),
                              iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*
        for i, element in zip(range(i+1, stop),
                              iterable):
            pass
```

→ from itertools import islice
print(list(islice('ABCDEFG', 2)))
print(list(islice('ABCDEFG', 2, 4)))
print(list(islice('ABCDEFG', 2, None)))



Page No. 683
Date

print(list(islice('ABCDEFG', 0, None, 2)))
Output- ['A', 'B']
['C', 'D']
['C', 'D', 'E', 'F', 'G']
['A', 'C', 'E', 'G']

(ix) starmap(function, iterable)- When an iterable is contained within another iterable and certain function has to be applied on them, starmap() is used. The ~~stoop~~ starmap() considers each element of the iterable within another iterable as a separate item. It is similar to map(). The difference between map() and starmap() parallels the ~~distribution~~ distinction between function(a,b) and function(*c).
Syntax- starmap(function, iterable).

Note- starmap() roughly equivalent to -

* def starmap(predicate, iterable)
    for args in iterable:
        yield function(*args)

→ from itertools import starmap
li= [(2,5), (3,2), (4,3)]
print(list(starmap(pow, li)))
print(list(starmap(lambda x,y: x+y, li)))
print(list(starmap(min, li)))
print(list(starmap(max, li)))

Page No. 684

Date

Output - [32, 9, 64]
[7, 5, 7]
[2, 2, 3]
[5, 3, 4]

(X) **takewhile(predicate, iterable)** - This iterator is opposite of dropwhile(). It prints the values till the function returns false for first (1st) time.

Syntax - takewhile(func, iterable)

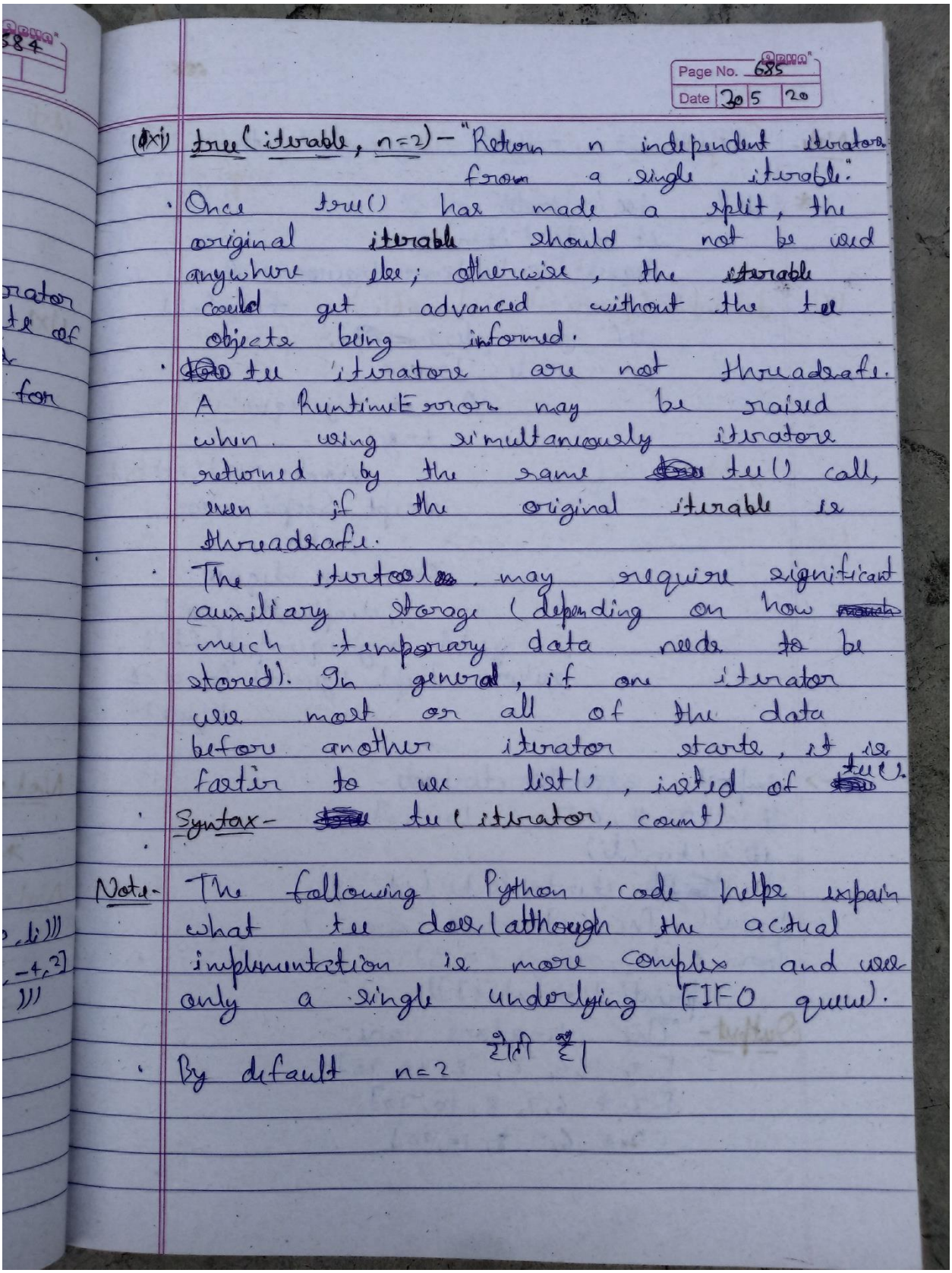
Note - takewhile() roughly equivalent to -

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

→

```
import itertools
li = [2, 4, 5, 7, 8]
print(list(itertools.takewhile(lambda x: x%2==0, li)))
print(list(itertools.takewhile(lambda x: x>0, [5, 6, -8, -4, 2]
)))
```

Output - [2, 4]
[5, 6]

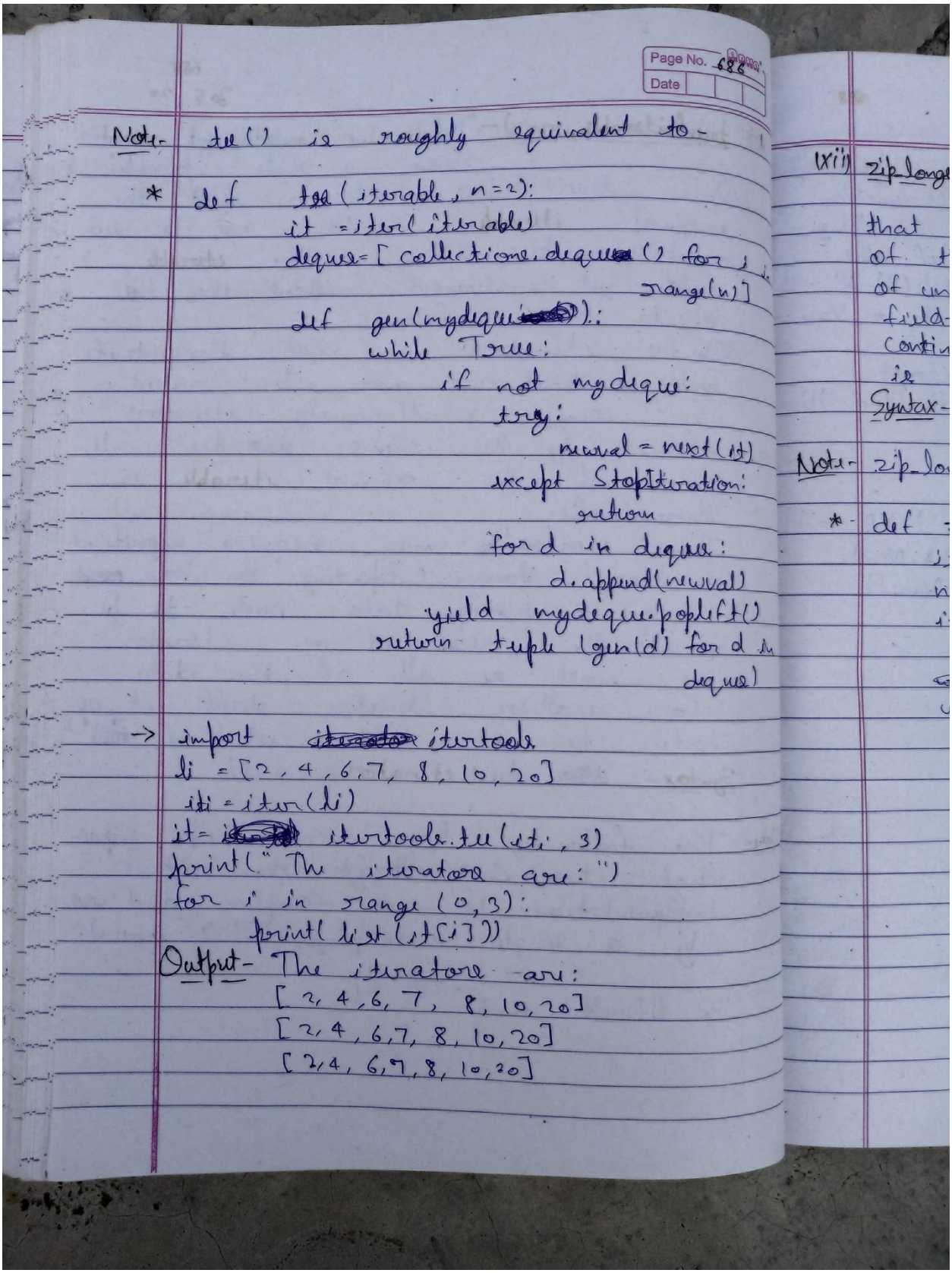


(xi) tee(iterable, n=2) – "Return n independent iterators from a single iterable."

- Once tee() has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed.
- tee iterators are not threadsafe. A RuntimeError may be raised when using simultaneously iterators returned by the same tee() call, even if the original iterable is threadsafe.
- The itertools may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use list(), instead of tee().
- Syntax- tee(iterator, count)

Note- The following Python code helps explain what tee does (although the actual implementation is more complex and uses only a single underlying FIFO queue).

- By default n=2 होती है।



Page No. 686
Date

Note- tee() is roughly equivalent to -

```
* def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in
                                range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:
                try:
                    newval = next(it)
                except StopIteration:
                    return
                for d in deques:
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in
                                deques)
```

```
-> import itertools
li = [2, 4, 6, 7, 8, 10, 20]
iti = iter(li)
it = itertools.tee(iti, 3)
print("The iterators are:")
for i in range(0, 3):
    print(list(it[i]))
```

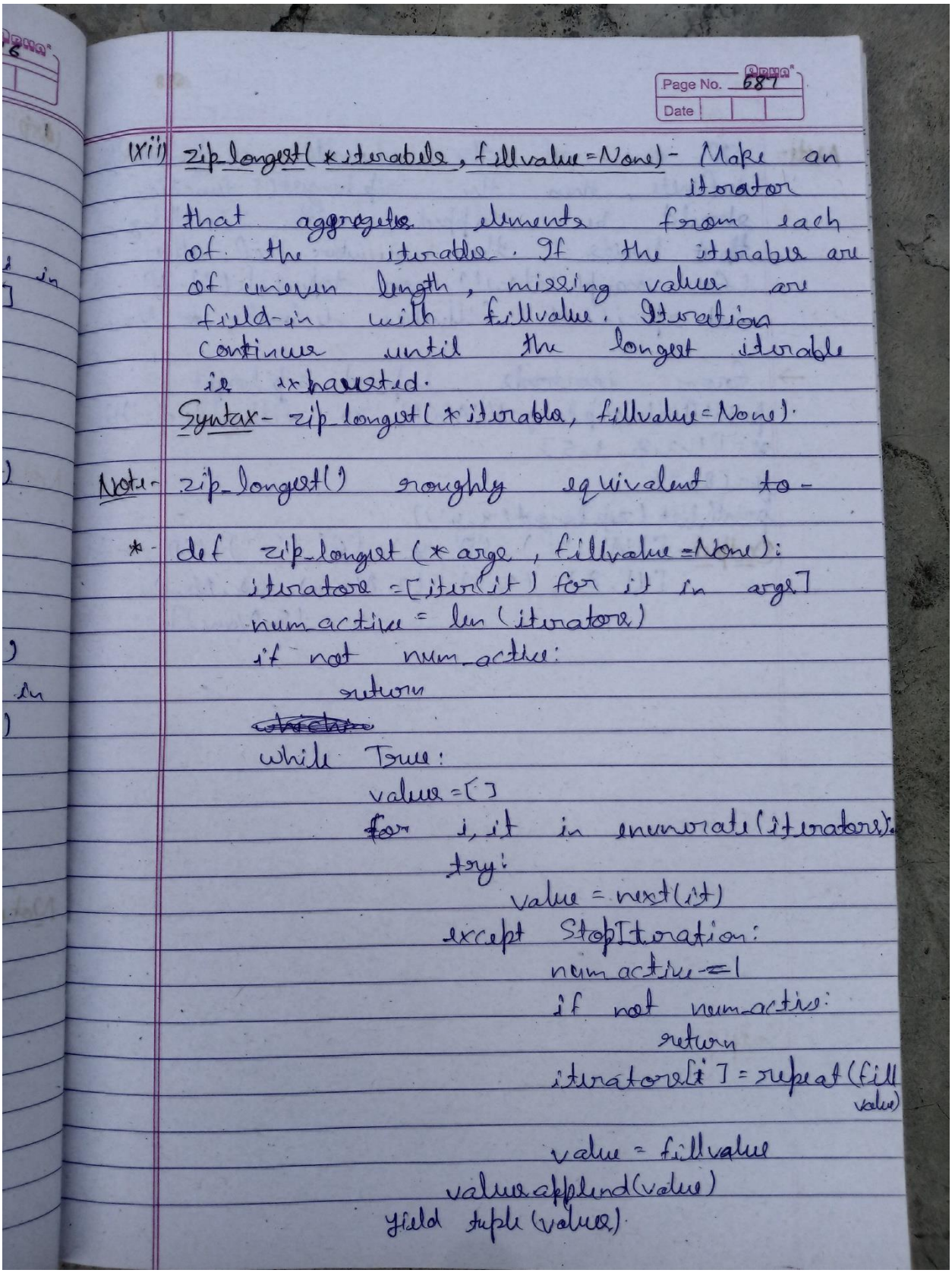
Output- The iterators are:
[2, 4, 6, 7, 8, 10, 20]
[2, 4, 6, 7, 8, 10, 20]
[2, 4, 6, 7, 8, 10, 20]

(xii) zip_longe

that
of t
of un
field
contin
is
Syntax-

Note- zip_lo

* def

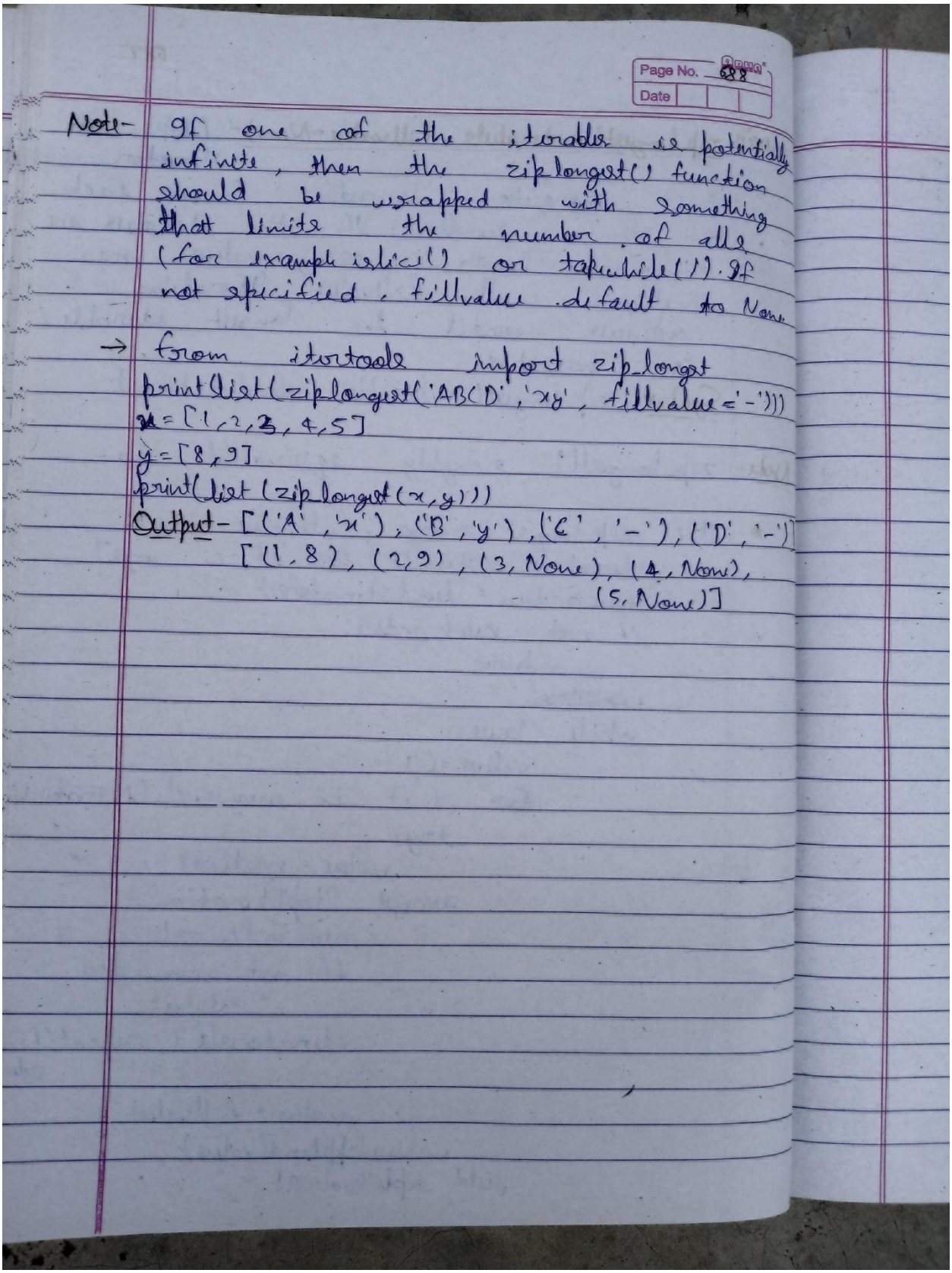


(xii) zip_longest(*iterables, fillvalue=None) - Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

Syntax - zip_longest(*iterable, fillvalue=None).

Note - zip_longest() roughly equivalent to -

```
def zip_longest(*args, fillvalue=None):
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```



Page No. 688
Date

Note- If one of the iterables is potentially infinite, then the zip_longest() function should be wrapped with something that limits the number of calls (for example islice() or takewhile()). If not specified, fillvalue defaults to None.

→ from itertools import zip_longest
print(list(zip_longest('ABCD', 'xy', fillvalue='-')))
x = [1, 2, 3, 4, 5]
y = [8, 9]
print(list(zip_longest(x, y)))
Output- [('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
[(1, 8), (2, 9), (3, None), (4, None), (5, None)]