PART-4

django Web Framework

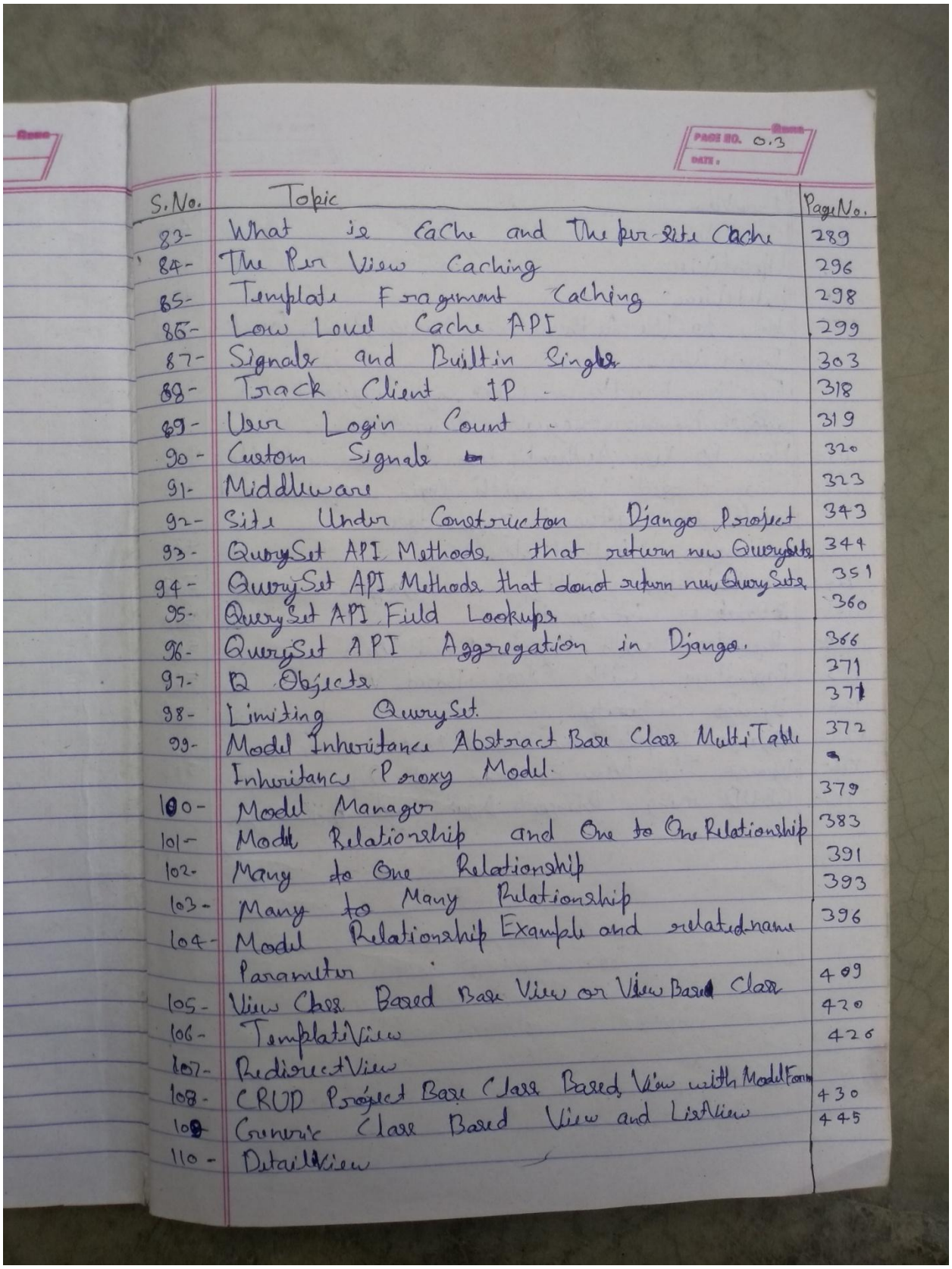
Geeky Shows YouTube Channel Learning Notes

SATYAM SETH

14/09/2020

Source Code – https://github.com/satyam-seth/django_learning

Playlist Link – https://www.youtube.com/playlist?list=PLbGui_ZYuhigchy8DTw4pX4duTTpvqlh6



PAGE NO. 0.3
DATE :

QuerySet API

PAGE NO. 344
DATE : 1-8-20

753 458 705 770 474 476 479 481 484 508 510 513 519 524 525

- A QuerySet can be defined as a list containing all those objects we have created using the Django model.
- Query property - This property is used to get sql query of query set.

Syntax - queryset.query

Methods that return new QuerySets

1- Retrieving all objects -

(i) all() - This method is used to retrieve all objects. This returns a copy of current QuerySet.

Syntax - Modlname.objects.all()

Ex - Student.objects.all()

2- Retrieving specific objects -

(i) filter(**kwargs) - It returns a new QuerySet containing objects that match the given lookup parameter. filter() will always give you a QuerySet, even if only a single object matches the query

Syntax - Modlname.objects.filter(Field = value)

Ex - Student.objects.filter(marks = 70)

PAGE NO. 345
DATE :

(ii) exclude(**kwargs) - It returns a new QuerySet containing objects that do not match the given lookup parameters.
Syntax- Modelname.objects.exclude(field = value)
Ex- Student.objects.exclude(marks=70)

(iii) order_by(*fields) - It orders the fields.

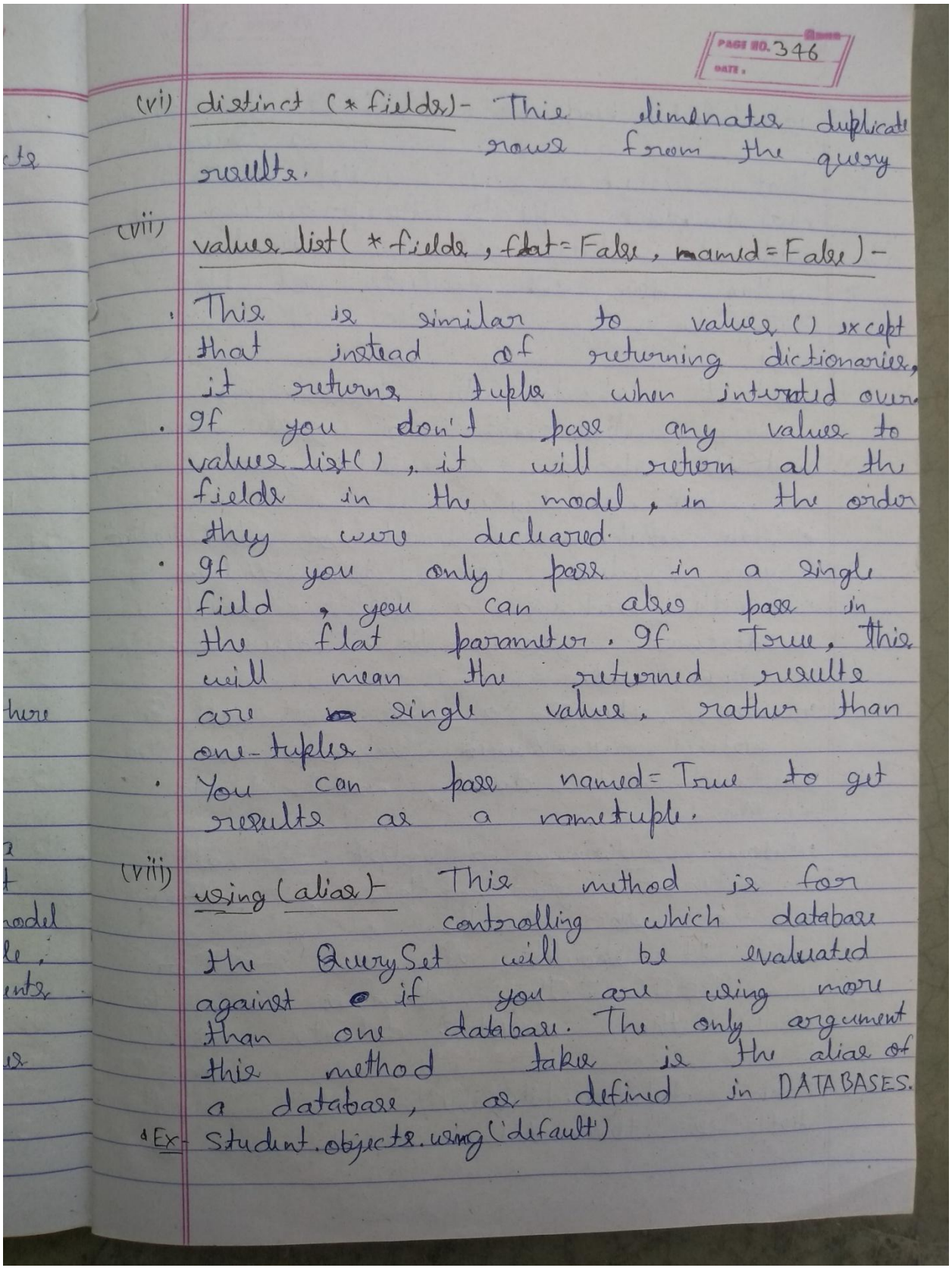
- 'field' - Asc order
- '-field' - Desc order
- '?' - Randomly

Ex- Student.objects.order_by("name")

Note- ये strings के data को unicode के अनुसार sort करता है। (A < a).

(iv) reverse() - This work only when there is ordering in queryset.
Ex- Student.objects.order_by('id').reverse()

(v) values(*fields, **expressions) - It returns a QuerySet that returns dictionaries, rather than model instances, when used as an iterable. Each of those dictionaries represents an object, with the keys corresponding to the attribute names of model objects.
Ex- Student.objects.values('name', 'city')



(vi) distinct (*fields)- This eliminates duplicate rows from the query results.

(vii) values_list(*fields, flat=False, named=False)-

- This is similar to values() except that instead of returning dictionaries, it returns tuple when iterated over.
- If you don't pass any values to values_list(), it will return all the fields in the model, in the order they were declared.
- If you only pass in a single field, you can also pass in the flat parameter. If True, this will mean the returned results are single values, rather than one-tuples.
- You can pass named=True to get results as a namedtuple.

(viii) using(alias)- This method is for controlling which database the QuerySet will be evaluated against if you are using more than one database. The only argument this method takes is the alias of a database, as defind in DATABASES.

Ex- Student.objects.using('default')

(ix) dates(field, kind, order='ASC') - It returns a QuerySet that evaluates to a list of datetime.date objects representing all available dates of a particular kind within the contents of the QuerySet.

Where,

- field - It should be the name of a DateField of your model.
- kind - It should be either "year", "month", "week", or "day".
  - "year" returns a list of all distinct year values for the field.
  - "month" returns a list of all distinct year/month values for the field.
  - "week" returns a list of all distinct year/week values for the field. All dates will be a Monday.
  - "day" returns a list of all distinct year/month/day values for the field.
- order - It should be either 'ASC' or 'DESC'. This specifies how to order the results. defaults to 'ASC'.

Note - Each datetime.date object in the result list is "truncated" to the given type.

(x) datetimes(field_name, kind, order='ASC', tzinfo=None)

It returns a QuerySet that evaluates to a list of datetime.datetime objects representing all available dates of a particular kind within the contents of the QuerySet.

- field_name - It should be the name of a DateTimeField of your model.
- kind - It should be either "year", "month", "week" or "day".
  - "year" returns a list of all distinct year values for the field.
  - "month" returns a list of all distinct year/month value for the field.
  - "week" returns a list of all distinct year/week values for the field. All dates will be a Monday.
  - "day" returns a list of all distinct year/month/day values for the field.
- order - It should be either 'ASC' or 'DESC'. This specifies how to order the results. defaults to 'ASC'.
- tzinfo - It defines the time zone to which datetimes are converted prior to truncation. This parameter must be a datatime.tzinfo object. If it's None, Django uses the current time zone.

PAGE NO. 349
DATE :

9t effect when USE_TZ is False.

Note: Each datetime.datetime object in the result list is "truncated" to the given type.

(xi) none()- Calling none() will create a queryset that never returns any objects and no query will be executed when accessing the results. A qs.none() queryset is an instance of EmptyQuerySet.

(xii) union(*other_qs, all=False)- Uses SQL's UNION operator to combine the results of two or more QuerySets. The UNION operator selects only distinct values by default. To allow duplicate values, use the all=True argument.

Ex- student_data = qs2.union(qs1, all=True)

(xiii) intersection(*other_qs)- Uses SQL's INTERSECT operator to return the shared elements of two or more QuerySets.

Ex- student_data = qs1.intersection(qs2)

(xiv) difference(*other_qs)- Uses SQL's EXCEPT operator to keep only elements present in the QuerySet but not in some other QuerySets.

Ex- student_data = qs1.difference(qs2)

PAGE NO. 350
DATE :

(xvi) select_related(*fields)
(xvii) defer(*fields)
(xviii) only(*fields)
(xix) prefetch_related(*lookups)
(xx) extra(select=None, where=None, params=None, tables=None, select_params=None)
(xxi) select_for_update(nowait=False, skip_locked=False, of=())
(xxii) raw(raw_query, params=None, translations=None)
(xxiii) annotate(*args, **kwargs)

Operators that return new QuerySets-

1- AND (&) - Combine two QuerySets using the SQL AND operator.

Ex- (i) student_data = Student.objects.filter(id=6) & Student.objects.filter(roll=106)

student_data = Student.objects.filter(id=6, roll=106)

student_data = Student.objects.filter(Q(id=6) & Q(roll=106))

2- OR (|) - Combine two QuerySets using the SQL OR operator.

Ex- Student.objects.filter(id=11) | Student.objects.filter(roll=106)

Student.objects.filter(Q(id=11) | Q(roll=106))

Note- Q को use करने से पहले import करना होगा।

from django.db.models import Q

99--

PAGE NO. 351
DATE: 2-8-20

Methods that do not return new QuerySets

1- Retrieving a single object-

(i) get() - It returns one single object. If there is no result match it will raise DoesNotExist exception. If more than one item matches the get() query. It will raise MultipleObjectReturned.

Ex- Student.objects.get(pk=1)

Note- यहाँ pk = primary key है इस अलावा हम कोई और field के साथ condition लगा सकते है बस वो records को uniquely identify कर पाये नहीं तो MultipleObjectReturned exception raise होगा।

. हम single object return करने वाले methods के साथ query का use नहीं कर सकते है

(ii) first() - It returns the first object matched by the queryset, or None if there is no matching object. If the QuerySet has no ordering defined, then the queryset is automatically ordered by the primary key.

Ex- (a) student_data = Student.objects.first()
(b) student_data = Student.objects.order_by('name').first()

PAGE NO. 352
DATE :

(iii) last() - It returns the last object matched by the queryset, or None if there is no matching object. If the QuerySet has no ordering defined, then the queryset is automatically ordered by the primary key.

Ex- (a) student_data = Student.objects.last()

(b) student_data = Student.objects.order_by('name').last()

(iv) latest(*fields) - It returns the latest object in the table based on the given field(s).

Ex- student_data = Student.objects.latest('pass_date')

(v) earliest(*fields) - It returns the earliest object in the table based on the given field(s).

Ex- student_data = Student.object.earliest('pass_date')

(vi) exists() - It returns True if the QuerySet contains any results, and False if not. This tries to perform the query in the simplest and fastest way possible, but it does execute nearly the same query as a normal QuerySet query.

Ex- student_data = Student.objects.all()
print(student_data.exists())

Note- इसका use करके हम ये पता लगा सकते हैं कि कोई record table में है या नहीं और फिर ना होने पर क्या है ये लिख सकते हैं।

PAGE NO. 353
DATE :

(vii) create(**kwargs) - A convenience method for creating an object and saving it all in one step.

Ex- s = Student.objects.create(name='Samuel', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')

Note- हमें इसी काम करने के लिये पहले 2 steps में करना पड़ता था -

s = Student(name='Samuel', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')
s.save(force_insert=True)

(viii) get_or_create(defaults=None, **kwargs) -

- A convenience method for looking up an object with the given kwargs (may be empty if your model has defaults for all fields), creating one if necessary.
- It returns a tuple of (object, created), where object is the retrieved or created and created is a boolean specifying whether a new object was created.

Ex- student_data, created = Student.objects.get_or_create(name='Samuel', roll=112, city='Bokaro', marks=60, pass_date='2020-5-4')
print(student_data, created)

(ix) update(**kwargs) - Performs an SQL update query for the specified fields, and returns the number of rows matched (which may not be equal

PAGE NO. 354
DATE :

to the number of rows updated if some rows already have the new value).

Ex- (a) student_data = Student.object.filter(id=12).update(name = 'Kabir', marks = 80)

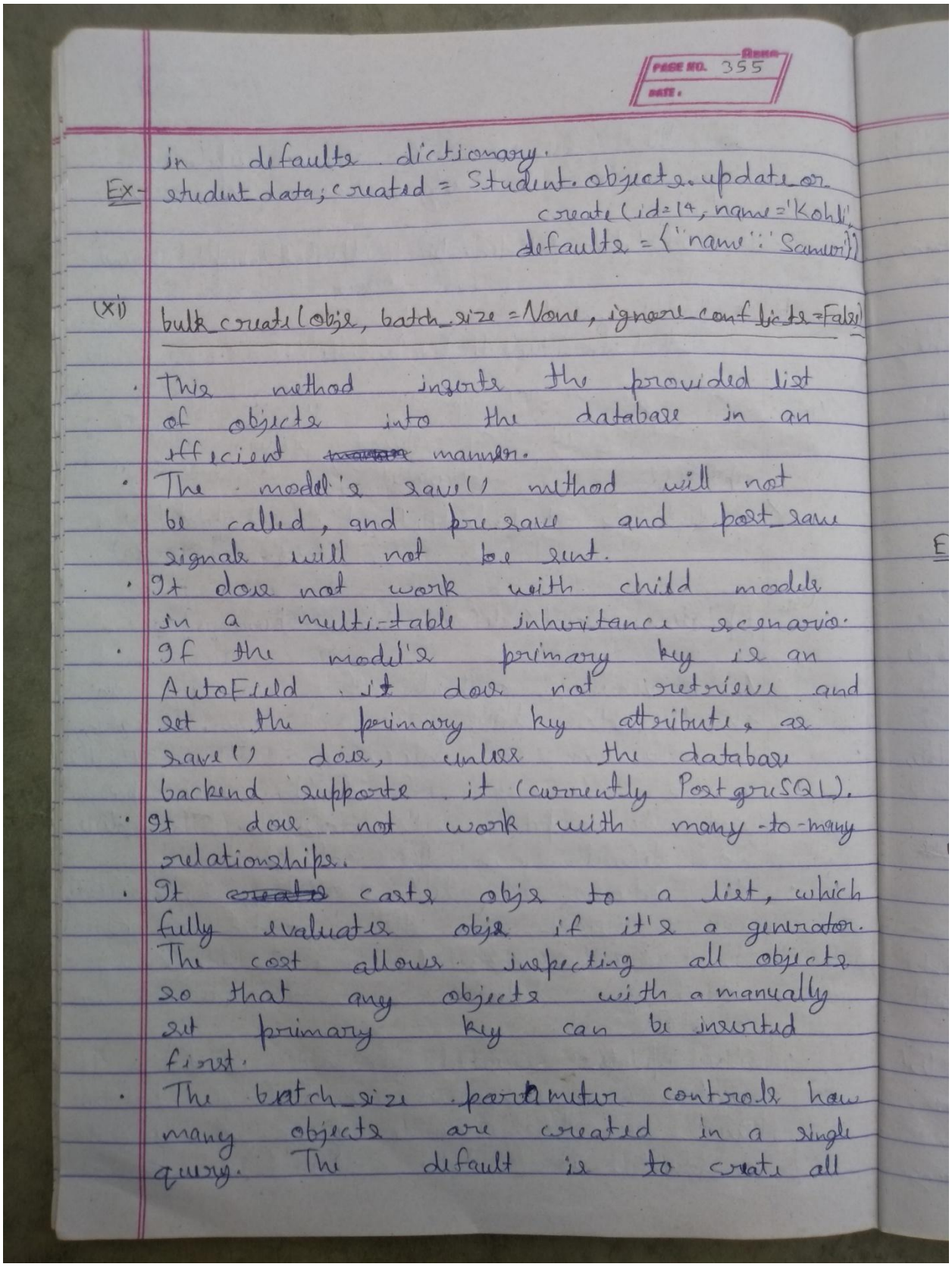
(b) # Update student's city Pass who has marks 60
student_data = Student.objects.filter(marks=60).update(city='Pass')

Note- हम get के साथ update का use नहीं कर सकते क्योंकि get एक object return करता है और update() को वो QuerySet पर ही काम करता है।

X- student_data = Student.objects.get(id=12).update(name = 'Kabir', marks = 80)

(X) update_or_create(defaults=None, **kwargs) -

- A convenience method for updating an object with the given kwargs, creating a new one if necessary. The defaults is a dictionary of (field, value) pairs used to update the object. The values in defaults can be callables.
- It returns a tuple of (object, created), where object is the created or updated object and created is a boolean specifying whether a new object was created.
- The update_or_created method tries to fetch an object from database based on the given kwargs. If a match is found, it updates the fields passed



in defaults dictionary.

Ex- student_data, created = Student.objects.update_or_create(id=14, name='Kohli', defaults={'name': 'Samuri'})

(xi) bulk_create(objs, batch_size=None, ignore_conflicts=False)

- This method inserts the provided list of objects into the database in an efficient manner.
- The model's save() method will not be called, and pre_save and post_save signals will not be sent.
- It does not work with child models in a multi-table inheritance scenario.
- If the model's primary key is an AutoField it does not retrieve and set the primary key attribute, as save() does, unless the database backend supports it (currently PostgreSQL).
- It does not work with many-to-many relationships.
- It casts objs to a list, which fully evaluates objs if it's a generator. The cast allows inspecting all objects so that any objects with a manually set primary key can be inserted first.
- The batch_size parameter controls how many objects are created in a single query. The default is to create all

PAGE NO. 356
DATE :

objects in one batch, except for SQLite where the default is such that at most 999 variables per query are used.

- On database that support it (all but Oracle), setting the ignore_conflicts parameter to True tells the database to ignore failure to insert any rows that fail constraints such as duplicate unique values. Enabling this parameter disables setting the primary key on each model instance.

Ex- objs = [
Student(name='Sonal', roll=120, city='Dhanbad', marks=40, pass_date='2020-5-4'),
Student(name='Kunal', roll=121, city='Dumka', marks=50, pass_date='20-5-7'),
Student(name='Aniza', roll=122, city='Giridih', marks=70, pass_date='2020-5-9')]
student_data = Student.objects.bulk_create(objs)

(xii) bulk_update(objs, fields, batch_size=None) -

- This method efficiently updates the given fields on the provided model instances, generally with one query. QuerySet.update() is used to save the changes, so this is more efficient than iterating through the list of methods and calling save() on each of them.
- You cannot update the model's primary key.

PAGE NO. 357
DATE :

- Each model's save() method isn't called, and the pre_save and post_save signals aren't sent.
- If updating a large number of columns in a large number of rows, the SQL generated can be very large. Avoid this by specifying a suitable batch_size.
- Updating fields defined on multi-table inheritance ancestors will incur an extra query per ancestor.
- If objs contains duplicates, only the first one is updated.
- The batch_size parameter controls how many objects are saved in a single query. The default is to update all objects in one batch, except for SQLite and Oracle which have restrictions on the number of variables used in a query.

Ex- all_student_data = Student.objects.all()
for stu in all_student_data:
    stu.city = 'Bhel'

student_data = Student.objects.bulk_update(all_student_data, ['city'])

(xiii) in_bulk(id_list = None, fieldname = 'pk') - It takes a list of field value (id_list) and the field_name for those values, and returns a dictionary mapping

each value to an instance of the object with the given field value. If id_list isn't provided, all objects in the queryset are returned. field_name must be a unique field, and it defaults to the primary key.

Ex- student_data = Student.objects.in_bulk([1,2])
print(student_data[1].name)
student_data1 = Student.objects.in_bulk()
print(student_data1)

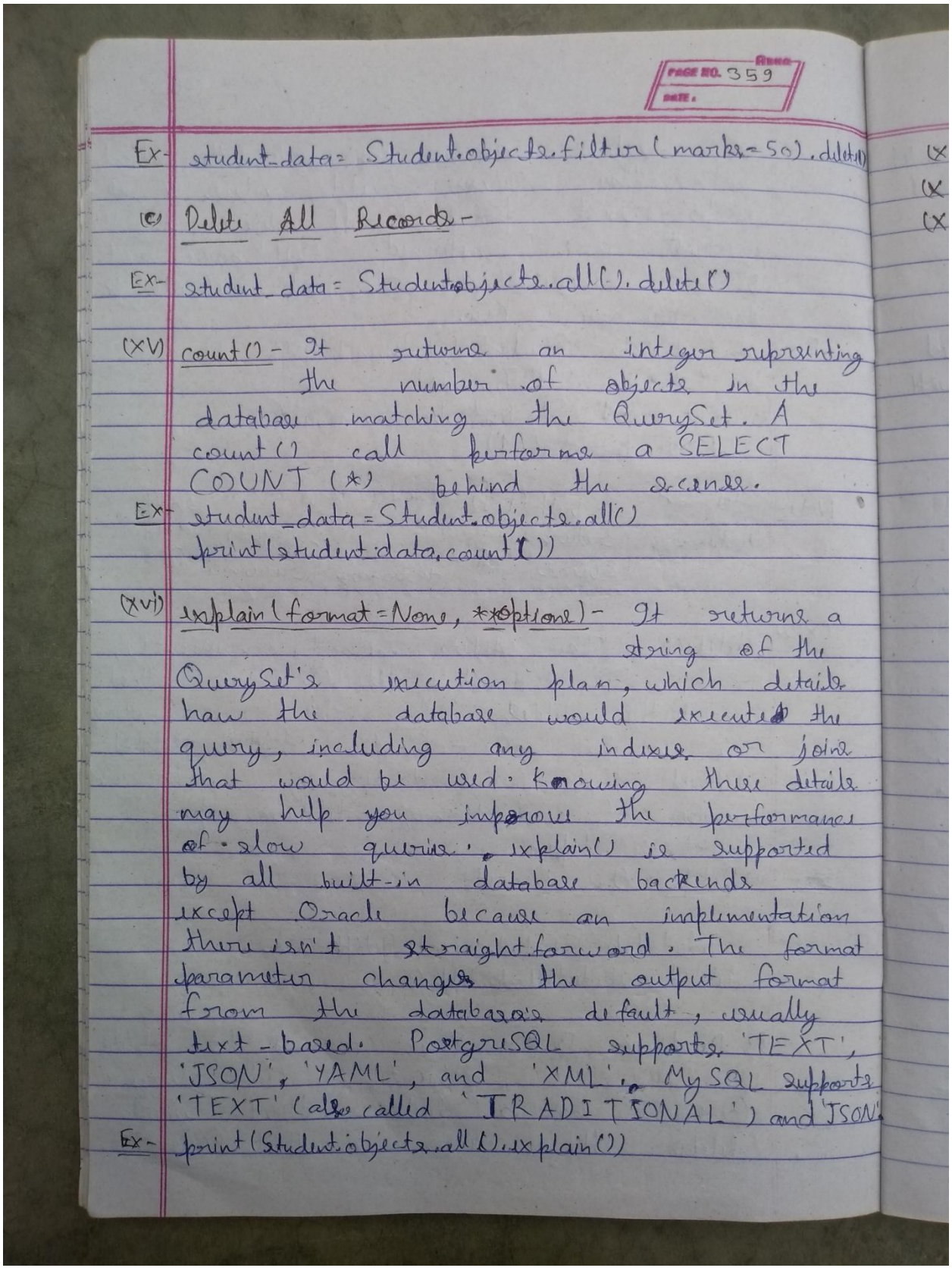
Note- जब हम in_bulk([]) लिखते है तो एक empty dictionary return होती है। और in_bulk() लिखते है तो सभी objects की dictionary return होती है।

(xiv) delete() - The delete method, conveniently, is named delete(). This method immediatly deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type.

- a) Delete One Record-

Ex- student_data = Student.objects.get(pk=22)
deleted = student_data.delete()

b) Delete in Bulk - You can also delete objects in bulk. Every QuerySet has a delete() method, which deletes all members of that QuerySet.



PAGE NO. 359
DATE :

Ex- student_data = Student.objects.filter(marks=50).delete()

(c) Delete All Records -

Ex- student_data = Student.objects.all().delete()

(xv) count() - It returns an integer representing the number of objects in the database matching the QuerySet. A count() call performs a SELECT COUNT(*) behind the scenes.

Ex- student_data = Student.objects.all()
print(student_data.count())

(xvi) explain(format=None, **options) - It returns a string of the QuerySet's execution plan, which details how the database would execute the query, including any indexes or joins that would be used. Knowing these details may help you improve the performance of slow queries. explain() is supported by all built-in database backends except Oracle because an implementation there isn't straightforward. The format parameter changes the output format from the database's default, usually text-based. PostgreSQL supports 'TEXT', 'JSON', 'YAML', and 'XML'. MySQL supports 'TEXT' (also called 'TRADITIONAL') and 'JSON'.

Ex- print(Student.objects.all().explain())

PAGE NO. 360
DATE: 3-8-20

(xvii) aggregate(*args, **kwargs)
(xviii) as_manager()
(xix) iterator(chunk_size=2000)

95

Field Lookups-

- Field lookups are how you specify the meat of an SQL WHERE clause.
- They are specified as keyword arguments to the QuerySet method filter(), ~~exclude()~~ exclude and get().
- If you pass an invalid keyword argument, a lookup function will raise TypeError.

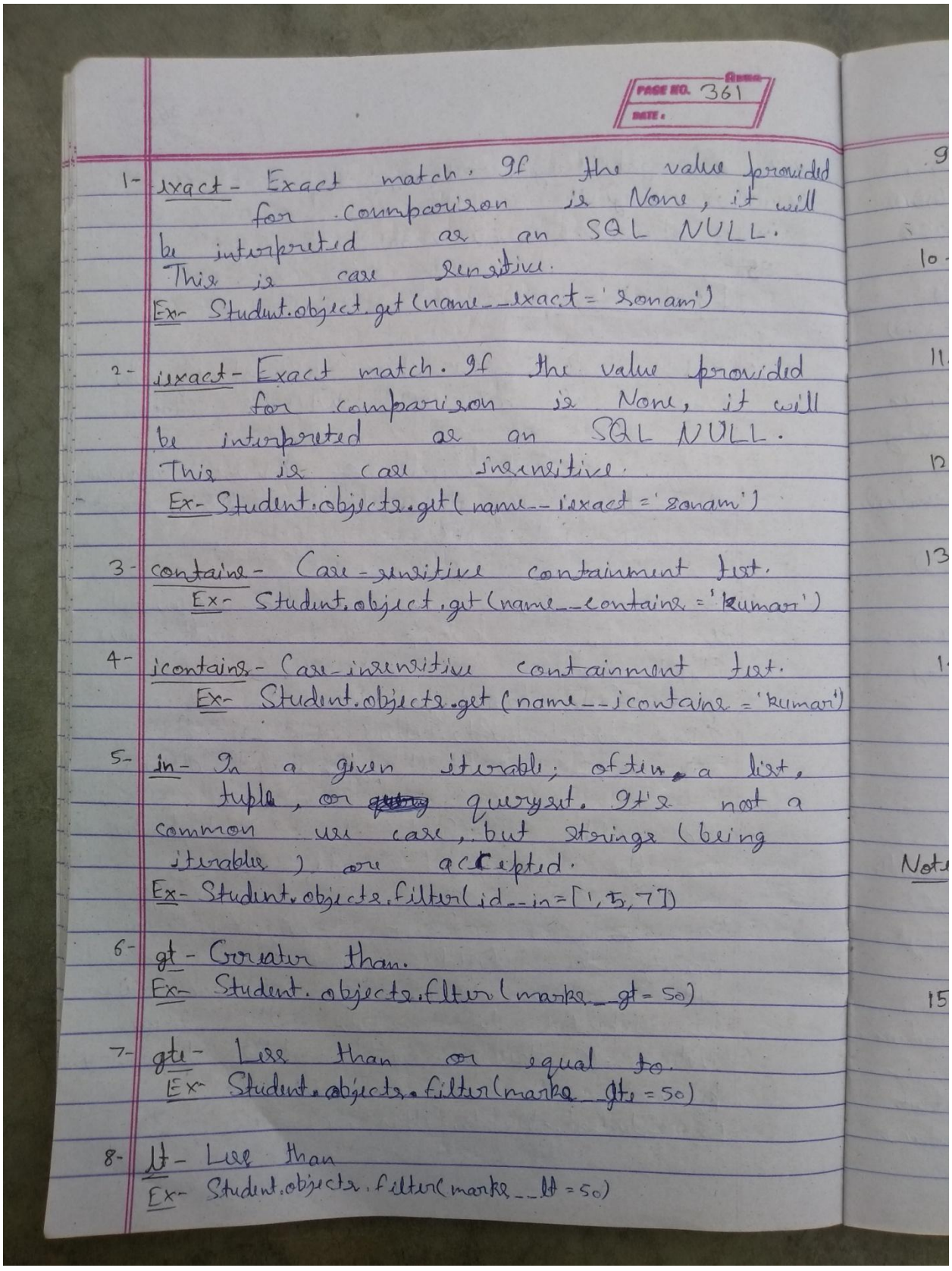
Syntax- field__lookuptype = value

Ex- Student.objects.filter(marks__lt='50')

SQL- SELECT * FROM myapp_student WHERE marks < '50';

- The field specified in a lookup has to be the name of model field.
- In ~~case~~ case of a ForeignKey you can specify the field name suffixed with _id. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key.

Ex- Student.objects.filter(stu_id=101



PAGE NO. 361
DATE :

1- iexact - Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL. This is case sensitive.
Ex- Student.object.get(name__iexact='sonam')

2- iexact - Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL. This is case insensitive.
Ex- Student.objects.get(name__iexact='sonam')

3- contains - Case-sensitive containment test.
Ex- Student.object.get(name__contains='kumar')

4- icontains - Case-insensitive containment test.
Ex- Student.objects.get(name__icontains='kumar')

5- in - In a given iterable; often a list, tuple, or ~~string~~ queryset. It's not a common use case, but strings (being iterable) are accepted.
Ex- Student.objects.filter(id__in=[1,5,7])

6- gt - Greater than.
Ex- Student.objects.filter(marks__gt=50)

7- gte - Less than or equal to.
Ex- Student.objects.filter(marks__gte=50)

8- lt - Less than
Ex- Student.objects.filter(marks__lt=50)

9- lte - Less than or equal to.
Ex- Student.objects.filter(marks__lte=50)

10- startswith - Case-sensitive starts-with.
Ex- Student.objects.filter(name__startswith='n')

11- istartswith - Case-insensitive starts-with.
Ex- Student.objects.filter(name__istartswith='n')

12- endswith - Case-sensitive ends-with.
Ex- Student.objects.filter(name__endswith='j')

13- iendswith - Case-insensitive ends-with.
Ex- Student.objects.filter(name__iendswith='j')

14- range - Range test (inclusive).
Ex- Student.objects.filter(pass_date_range = ('2020-04-01','2020-05-05'))

SQL - SELECT ... WHERE admission_date BETWEEN '2020-04-01' and '2020-05-05';

Note- You can use range anywhere you can use BETWEEN in SQL - for dates, numbers and even characters.

15- date - For datetime fields, casts the value as date. Allows chaining field lookups. Takes a date value.
Ex- (i) Student.objects.filter(admdatetime__date=date(2020,6,5))

(ii) Student.objects.filter(admdatetime__date__gt=date(2020,6,5))

PAGE NO. 363
DATE :

16- year- For date and datetime fields, an exact year match. Allows chaining additional field lookups. Takes an integer year.
Ex- (i) Student.objects.filter(pass_date__year=2020)
(ii) Student.objects.filter(pass_date__year__gt=2019)

17- month- For date and datetime fields, an exact month match. Allows chaining additional field lookups. Takes an integer 1 (January) through 12 (December).
Ex- (i) Student.objects.filter(pass_date__month=6)
(ii) Student.objects.filter(pass_date__month__gt=5)

18- day- For date and datetime fields, an exact day match. Allows chaining additional field lookups. Takes an integer day.
Ex- (i) Student.objects.filter(pass_date__day=5)
(ii) Student.objects.filter(pass_date__day__gt=3)

Note- This will match any record with a pub_date on the third day of the month, such as January 3, July 3 etc.

19- week- For date and datetime fields, return the week number (1-52 or 53) according to ISO-8601, i.e., weeks start on a Monday and the first week contains the year's first Thursday.

PAGE NO. 364
DATE :

Ex- (i) Student.objects.filter(pass date__week=23)
(ii) Student.objects.filter(passdate__week__gt=22)

20- week_day - For date and datetime fields, a day of the week match. Allows chaining additional field lookups. Take an integer value representing the day of week from 1 (Sunday) to 7 (Saturday).
Ex- Student.objects.filter(pass date__week_day=6)
Student.objects.filter(passdate__week_day__gt=5)

Note- This will match any record with a admission_date that falls on a Monday (day = 2 of the week), regardless of the month or year in which it occurs. Week days are indexed with day 1 being Sunday and day 7 being Saturday.

21- quarter - For date and datetime fields, a 'quarter of the year' match. Allows chaining additional field lookups. Take an integer value between 1 and 4 representing the quarter of the year.
Example to retrieve entries in the second quarter (April 1 to June 30.)-
Ex- Student.objects.filter(passdate__quarter=2)

PAGE NO. 365
DATE :

22- time- For datetime fields, casts the value as time. Allows chaining additional field lookups. Takes a datetime.time value.
Ex- Student.objects.filter(admdatetime__time__gt=time(6,00))

23- hour- For datetime and time fields, an exact hour match. Allows chaining additional field lookups. Takes an integer between 0 and 23.
Ex- Student.objects.filter(admdatetime__hour__gt=5)

24- minute- For datetime and time fields, an exact minute. Allows chaining additional field lookups. Takes an integer between 0 and 59.

25- second- For datetime and time fields, an exact second match. Allows chaining additional field lookups. Takes an integer between 0 and 59.
Ex- Student.objects.filter(admdatetime__second__gt=30)

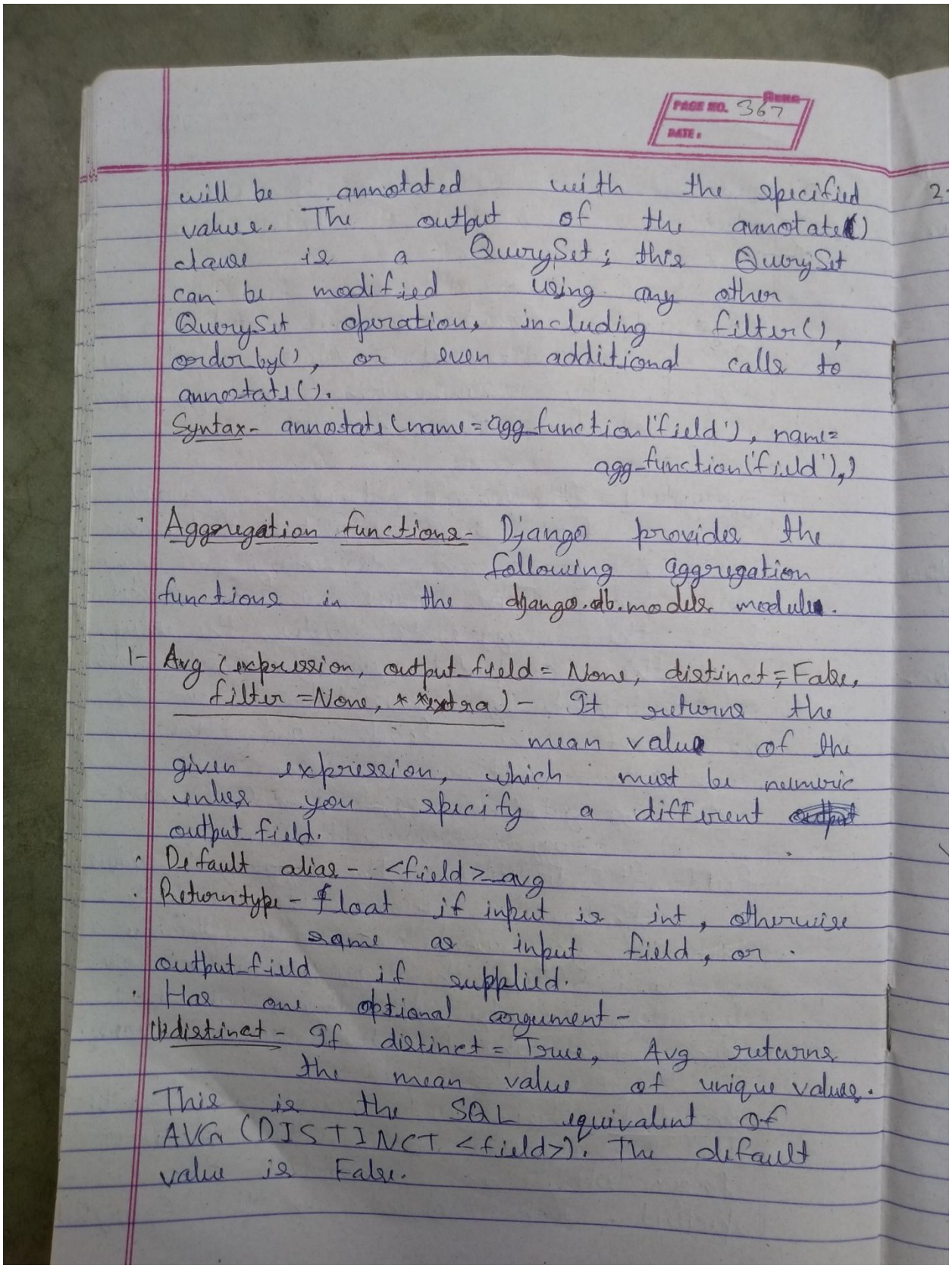
26- isnull- Takes either True or False, which correspond to SQL queries of IS NULL and IS NOT NULL, respectively.
Ex- Student.objects.filter(roll__isnull=False)

PAGE NO. 356
DATE :

27- regex
28- iregex
98 —

Aggregation- Sometimes you need to retrieve values that are devided by summarizing or aggregating a collection of objects.

- aggregate() - It is a terminal clause for a QuerySet that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function.
  Syntax aggregate(name = agg_function('field'), name = agg_function('field'),)
- field- It describes the aggregate value that we want to compute.
- name- If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause.
- annotate() - Per-object summaries can be generated using the annotate() clause. When an annotate() clause is specified, each object in the QuerySet



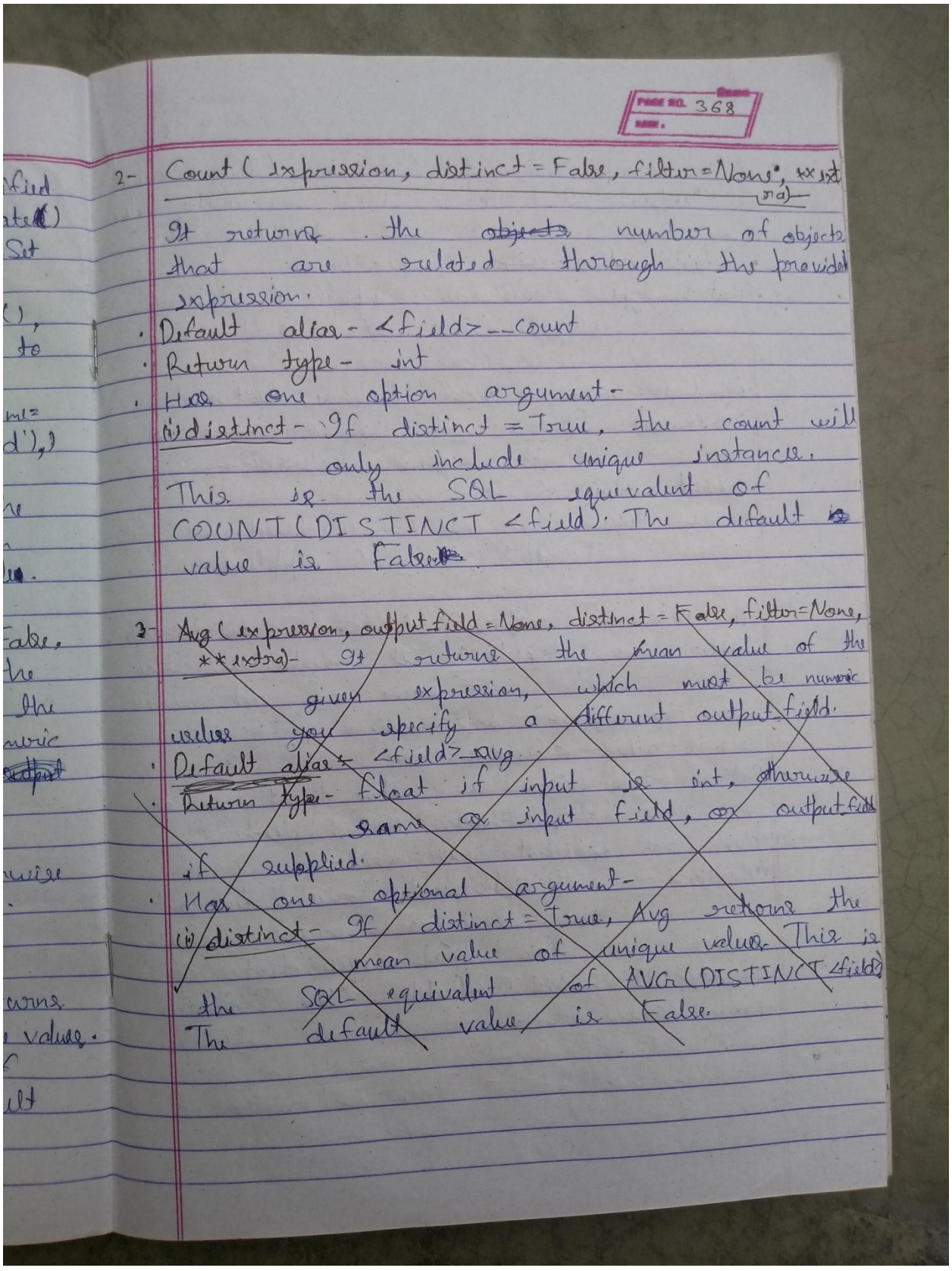
will be annotated with the specified values. The output of the annotate() clause is a QuerySet; this QuerySet can be modified using any other QuerySet operation, including filter(), order_by(), or even additional calls to annotate().

Syntax- annotate(name=agg_function('field'), name= agg_function('field'),)

- Aggregation functions- Django provides the following aggregation functions in the django.db.models module.

1- Avg(expression, output_field=None, distinct=False, filter=None, **extra)- It returns the mean value of the given expression, which must be numeric unless you specify a different output_field.
- Default alias- <field>_avg
- Return type- float if input is int, otherwise same as input field, or output_field if supplied.
- Has one optional argument-

distinct- If distinct=True, Avg returns the mean value of unique values. This is the SQL equivalent of AVG(DISTINCT <field>). The default value is False.



PAGE NO. 368

2- Count(expression, distinct = False, filter = None, **extra)

It returns the ~~objects~~ number of objects that are related through the provided expression.

- Default alias - <field>__count
- Return type - int
- Has one option argument -

(i) distinct - If distinct = True, the count will only include unique instances. This is the SQL equivalent of COUNT(DISTINCT <field>). The default value is False.

3- Avg(expression, output_field = None, distinct = False, filter = None, **extra) - It returns the mean value of the given expression, which must be numeric unless you specify a different output_field.

- Default alias = <field>__avg
- Return type - float if input is int, otherwise same as input field, or output_field if supplied.
- Has one optional argument -

(i) distinct - If distinct = True, Avg returns the mean value of unique values. This is the SQL equivalent of AVG(DISTINCT <field>). The default value is False.

PAGE NO. 369
DATE: 4-8-20

3- Max(expression, output_field=None, filter=None, **extra) - It returns the maximum value of the given expression.
- Default alias - <field>__max
- Return type - same as input field, or output field if supplied.

4- Min(expression, output_field=None, filter=None, **extra) - It returns the maximum value of the given expression.
- Default alias - <field>__min
- Return type - same as input field, or output_field if supplied.

5- Sum(expression, output_field=None, distinct=False, filter=None, **extra) - It computes the sum of all values of the given expression.
- Default alias - <field>__sum
- Return type - same as input field, or output_field if supplied.
- Has one optional argument -
  (i) distinct - If distinct=True, Sum returns the sum of unique values. This is the SQL equivalent of SUM(DISTINCT <field>). The default value is False.

6- StdDev(expression, output_field=None, sample=False, filter=None, **extra) - It returns the standard derivation of the data in the provided expression.

- Default alias - <field>--stddev
- Return type - float if input is int, otherwise same as input field, or output_field if supplied.
- Has one optional argument -
  (i) sample - By default, StdDev returns the population standard deviation. However, if sample=True, the return value will be sample standard deviation.

7- Variance(expression, output_field=None, sample=False, filter=None, **extra) - It returns the variance of the data in the provided expression.

- Default alias - <field>--variance
- Return type - float if output is int, otherwise same as input field, or output_field if supplied.
- Has one optional argument -
  (i) sample - By default, Variance returns the population variance. However, if sample=True, the return value will be the sample variance.

97 -

PAGE NO. 371
DATE : 5-8-20

Q Objects-

- Q object is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in "Field lookups".
- If you need to execute more complex queries, you can use Q objects.
- Q objects can be combined using the & and | operators. When an operator is used on two Q objects, it yields a new Q object.

from django.db.models import Q.

1- & (AND) Operator-

Ex- Student.object.filter(Q(id=6) & Q(roll=106))

2- | (OR) Operator-

Ex- Student.objects.filter(Q(id=6) | Q(roll=100))

3- ~ (Negation) Operator-

Ex- Student.objects.filter(~Q(id=6))

99 —

Limiting QuerySets - Use a subset of python's array-slicing syntax to limit your QuerySet to a certain number of results. This is the equivalent of SQL's LIMIT and OFFSET clause.

Ex- (i) Studentobjects.all()[:5] - This returns First 5 objects.

PAGE NO. 372
DATE: 5-8-20

(ii) Student.objects.all()[5:10] - This returns sixth through tenth objects.

(iii) Student.objects.all()[-1] → This is not valid.

(iv) Student.objects.all()[:10:2] - This returns a list of every second object of the first 10.

Note- step की value देने पर QuerySet के बजाय list return होता है इसलिए हम इसके साथ .query attribute का use नहीं कर सकते है।

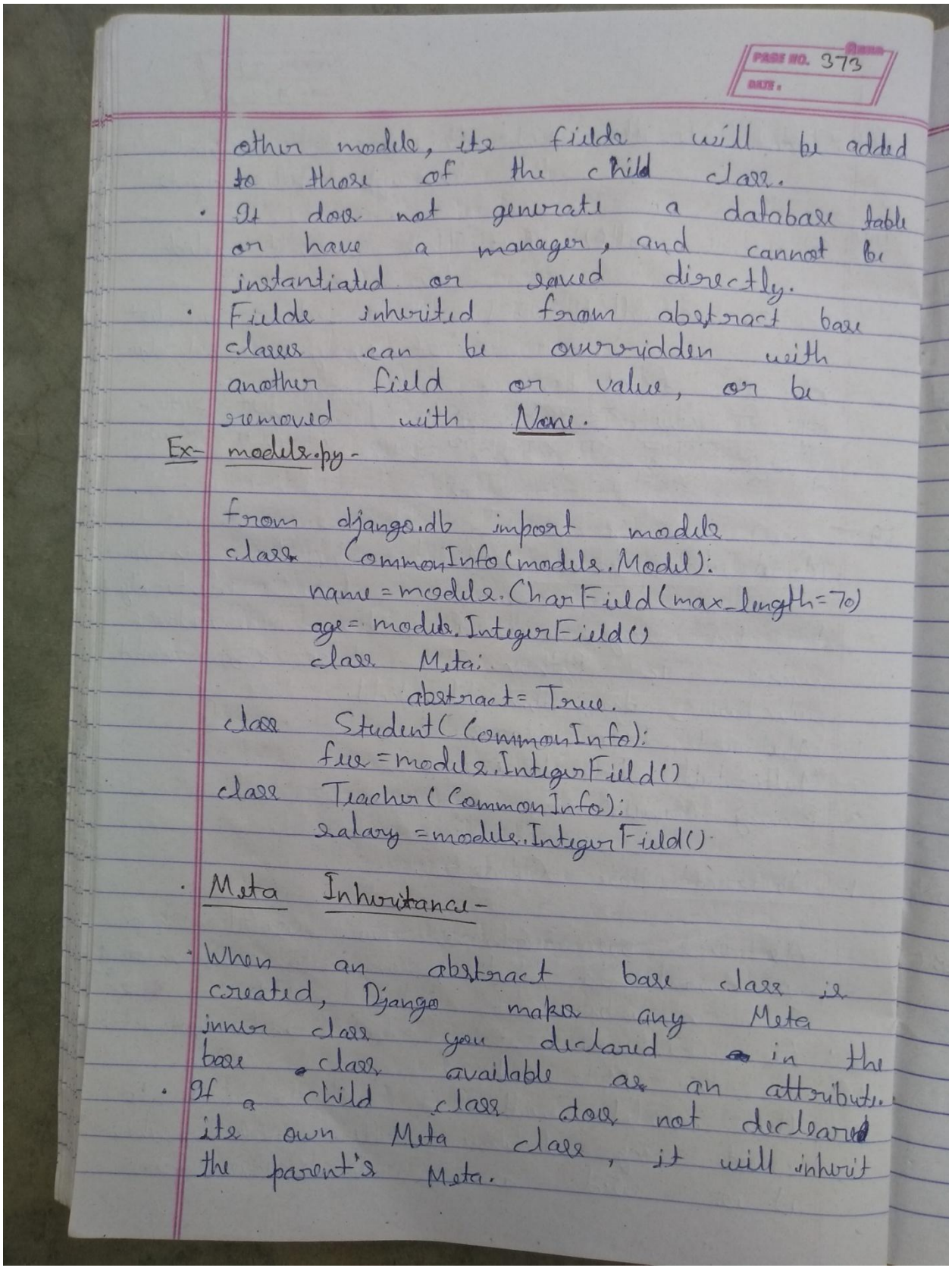
99 -

Model Inheritance - Model inheritance in Django works almost identically to the way normal class inheritance works in Python. The base class should subclass django.db.models.Model.

1- Abstract Base Classes
2- Multi-table Inheritance
3- Proxy Models.

1- Abstract Base Class -

- Abstract base classes are useful when you want to put some common information into a number of other models.
- You write your base class and abstract=True in the Meta class.
- This model will then not be used to create any database table. Instead, when it is used as a base class for



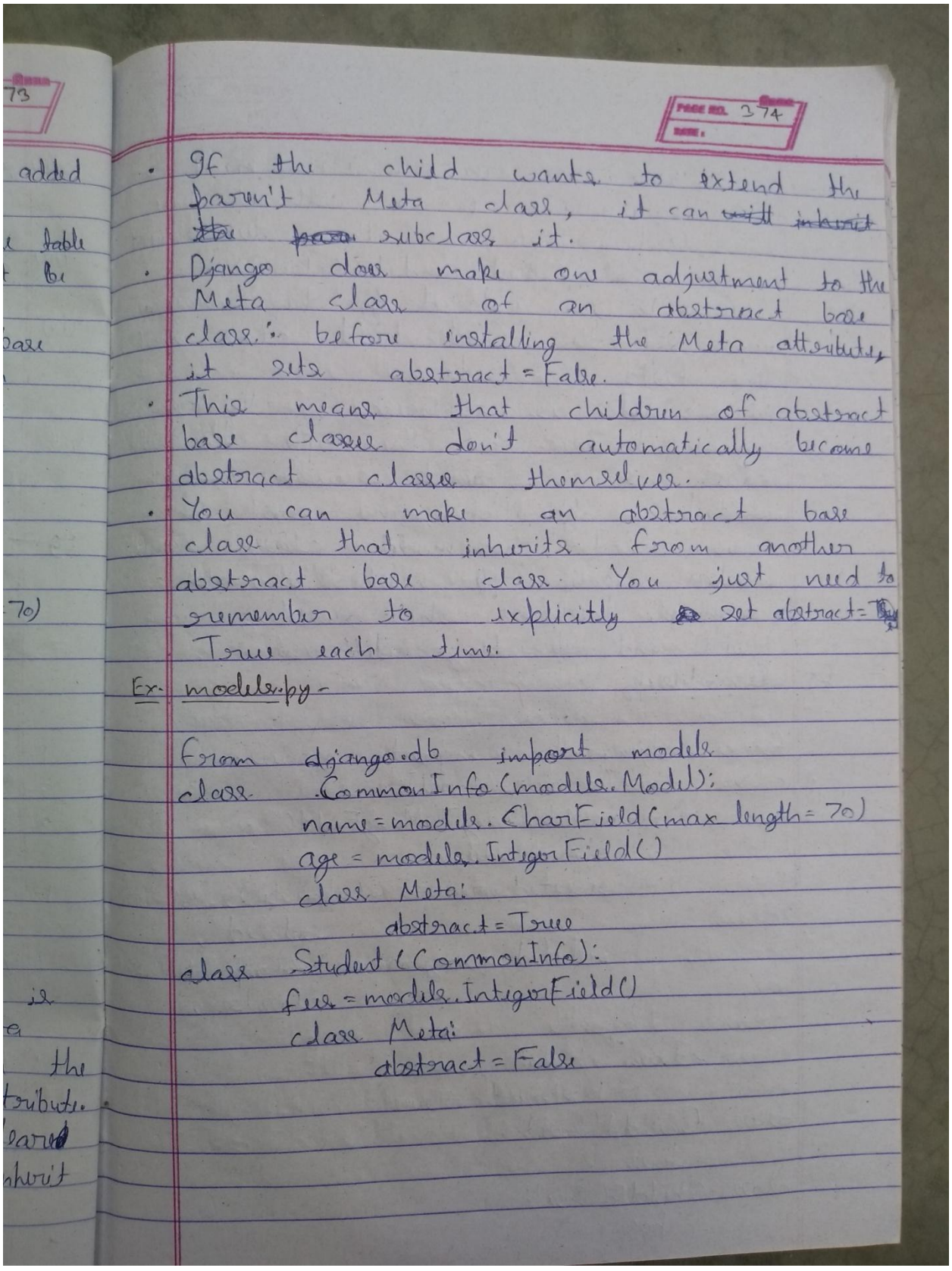
other module, its fields will be added to those of the child class.
- It does not generate a database table or have a manager, and cannot be instantiated or saved directly.
- Fields inherited from abstract base classes can be overridden with another field or value, or be removed with None.

Ex- models.py -

```
from django.db import models
class CommonInfo(models.Model):
    name = models.CharField(max_length=70)
    age = models.IntegerField()
    class Meta:
        abstract = True.
class Student(CommonInfo):
    fee = models.IntegerField()
class Teacher(CommonInfo):
    salary = models.IntegerField()
```

- Meta Inheritance -

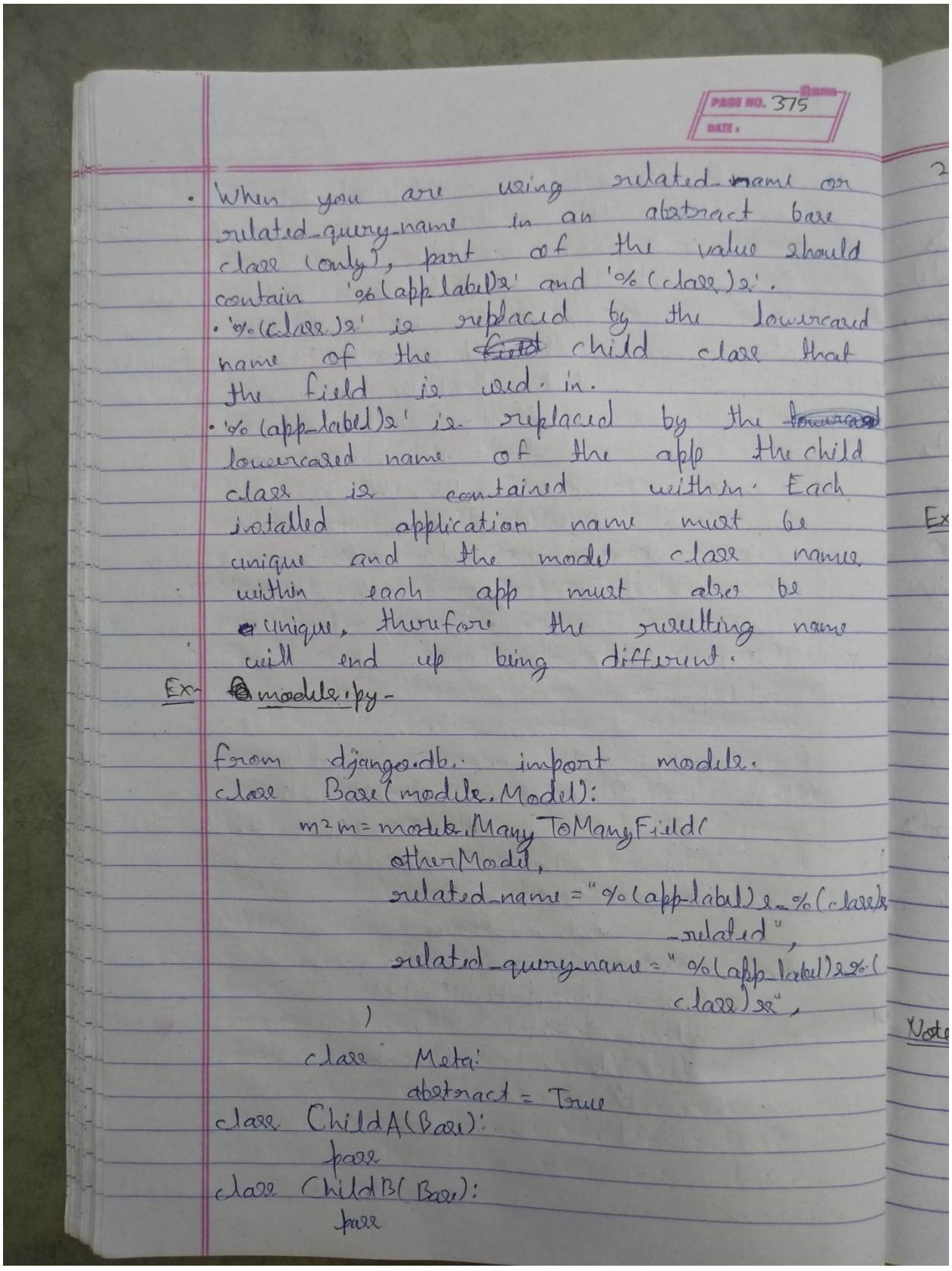
- When an abstract base class is created, Django makes any Meta inner class you declared in the base class available as an attribute.
- If a child class does not declare its own Meta class, it will inherit the parent's Meta.



- If the child wants to extend the parent's Meta class, it can ~~with inherit the parent~~ subclass it.
- Django does make one adjustment to the Meta class of an abstract base class: before installing the Meta attribute, it sets abstract = False.
- This means that children of abstract base classes don't automatically become abstract classes themselves.
- You can make an abstract base class that inherits from another abstract base class. You just need to remember to explicitly set abstract = True each time.

Ex. models.py -

```
From django.db import models
class CommonInfo(models.Model):
    name = models.CharField(max_length=70)
    age = models.IntegerField()
    class Meta:
        abstract = True
class Student(CommonInfo):
    fee = models.IntegerField()
    class Meta:
        abstract = False
```

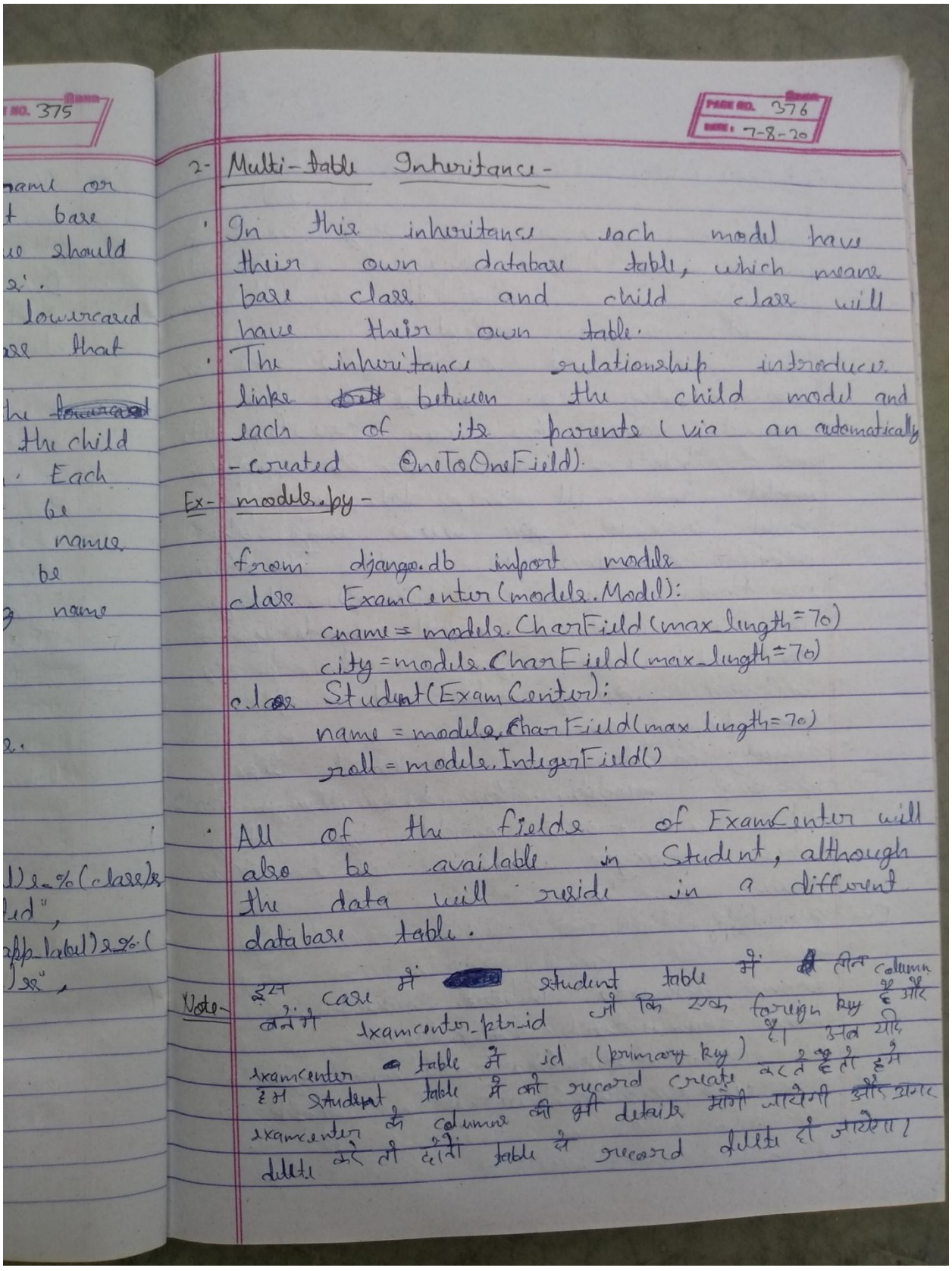


PAGE NO. 375
DATE :

- When you are using related_name or related_query_name in an abstract base class (only), part of the value should contain '%(app_label)s' and '%(class)s'.
- '%(class)s' is replaced by the lowercased name of the ~~field~~ child class that the field is used in.
- '%(app_label)s' is replaced by the ~~lowercased~~ lowercased name of the app the child class is contained within. Each installed application name must be unique and the model class name within each app must also be unique, therefore the resulting name will end up being different.

Ex- module.py -

```
from django.db import models
class Base(models.Model):
    m2m = models.ManyToManyField(
        otherModel,
        related_name = "%(app_label)s_%(class)s_related",
        related_query_name = "%(app_label)s_%(class)s",
    )
    class Meta:
        abstract = True
class ChildA(Base):
    pass
class ChildB(Base):
    pass
```



PAGE NO. 376
DATE: 7-8-20

2- Multi-table Inheritance -

- In this inheritance each model have their own database table, which means base class and child class will have their own table.
- The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created OneToOneField).

Ex- models.py -

```
from django.db import models
class ExamCenter(models.Model):
    cname = models.CharField(max_length=70)
    city = models.CharField(max_length=70)
class Student(ExamCenter):
    name = models.CharField(max_length=70)
    roll = models.IntegerField()
```

- All of the fields of ExamCenter will also be available in Student, although the data will reside in a different database table.

Note- इस case में student table में तीन columns बनेंगे examcenter_ptr_id जो कि एक foreign key है और examcenter table में id (primary key) है। अब यदि हम student table में कोई record create करते हैं तो हमें examcenter के columns की भी details माँगी जायेगी और अगर delete करें तो दोनों table से record delete हो जायेगा।

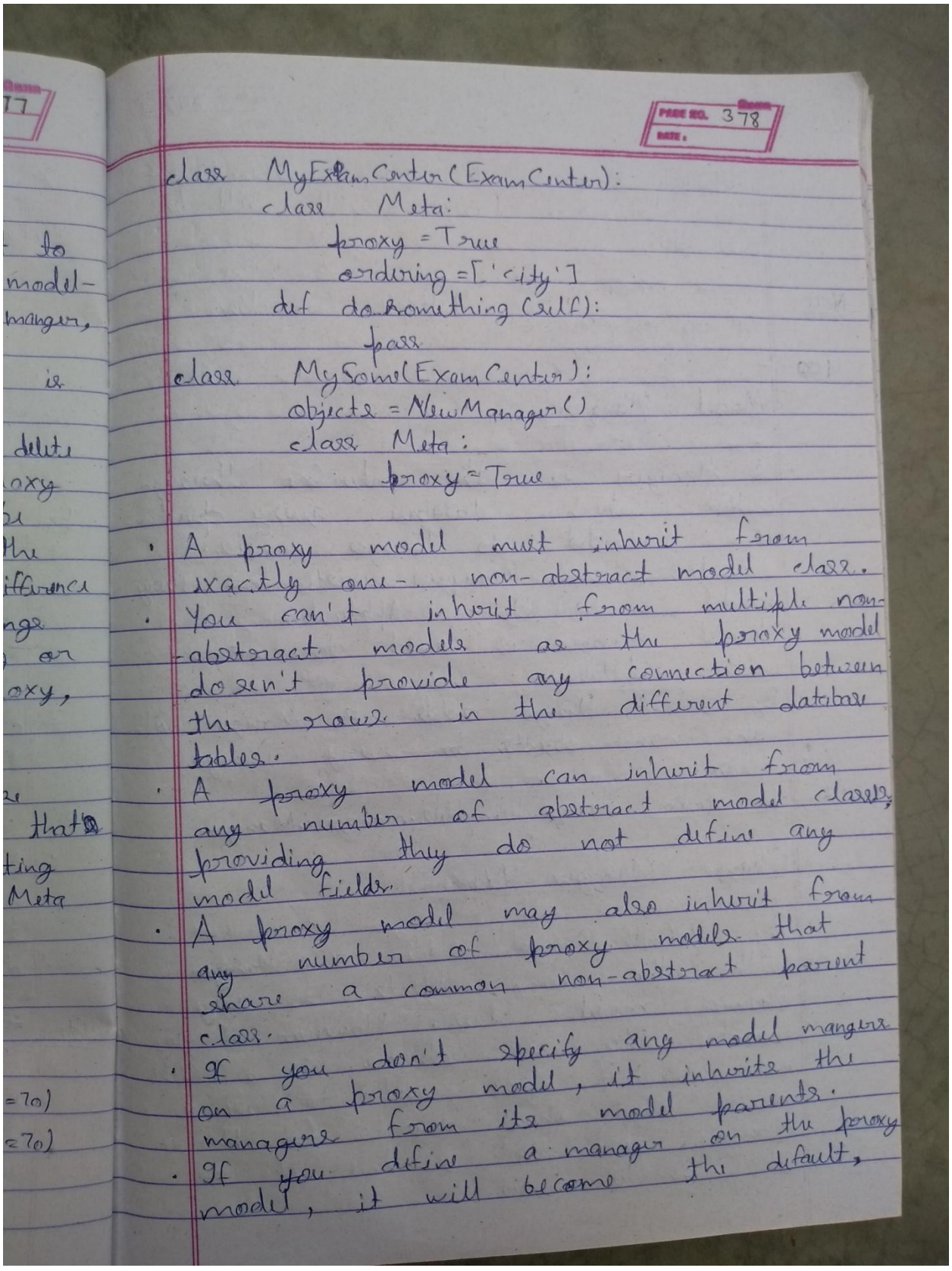
PAGE NO. 377
DATE :

3- Proxy Model-

- Sometimes, however, you only want to change the Python behavior of a model- perhaps to change the default manager, or add a new method.
- This is what model inheritance is for: creating a proxy for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model. The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original.
- Proxy models are declared like normal models. You tell Django that it's a proxy model by setting the proxy attribute of the Meta class to True.

Ex- models.py-

```
from django.db import models
class ExamCenter(models.Model):
    cname = models.CharField(max_length=70)
    city = models.CharField(max_length=70)
```



```
class MyExamCenter(ExamCenter):
    class Meta:
        proxy = True
        ordering = ['city']
    def do_something(self):
        pass
class MySomeExamCenter:
    objects = NewManager()
    class Meta:
        proxy = True
```

- A proxy model must inherit from exactly one non-abstract model class.
- You can't inherit from multiple non-abstract models as the proxy model doesn't provide any connection between the rows in the different database tables.
- A proxy model can inherit from any number of abstract model classes, providing they do not define any model fields.
- A proxy model may also inherit from any number of proxy models that share a common non-abstract parent class.
- If you don't specify any model managers on a proxy model, it inherits the managers from its model parents.
- If you define a manager on the proxy model, it will become the default,

although any managers defined on the parent classes will still be available.

Ex-

Note- इसके अलावा हम Multiple inheritance का भी use कर सकते है।

100 —

Model Manager-

- A Manager is the interface through ~~which~~ which database query operations are provided to Django models.
- At least one Manager exists for every model in a Django Application.
- Model manager is used to interact with database.
- By default, Django adds a Manger with the name objects to every Django class.

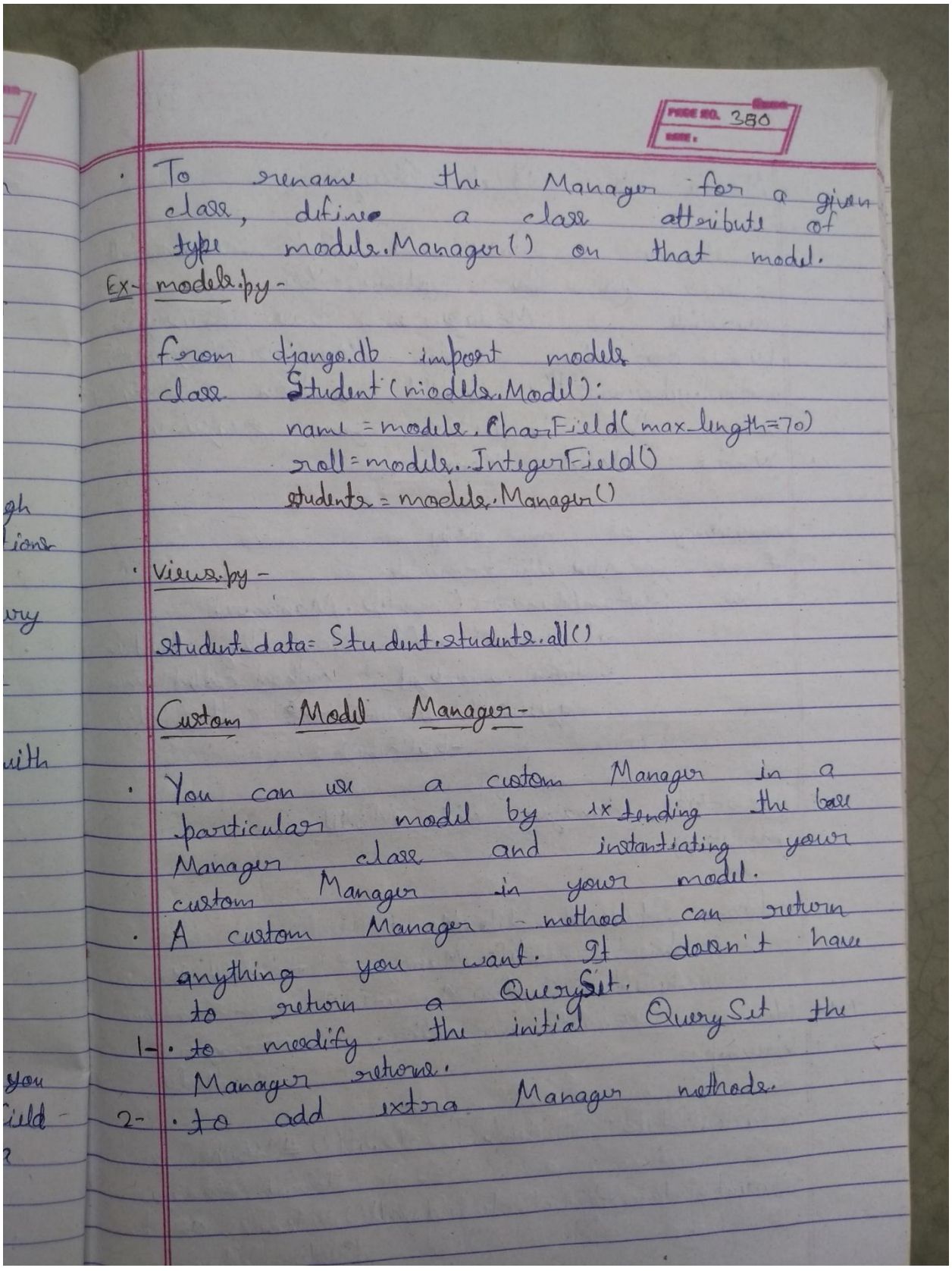
Available at location -- django.db.models.manager.Manager

Change Manager Name-

- By default, Django adds a Manager with the name objects to every Django model class. However, if you want to use objects as a field name, or if you want to use a name other than objects for the Manager, you can rename it on a per-model basis.

1-

2-



PAGE NO. 380
DATE :

- To rename the Manager for a given class, define a class attribute of type models.Manager() on that model.

Ex- models.py -

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=70)
    roll = models.IntegerField()
    students = models.Manager()
```

- views.py -

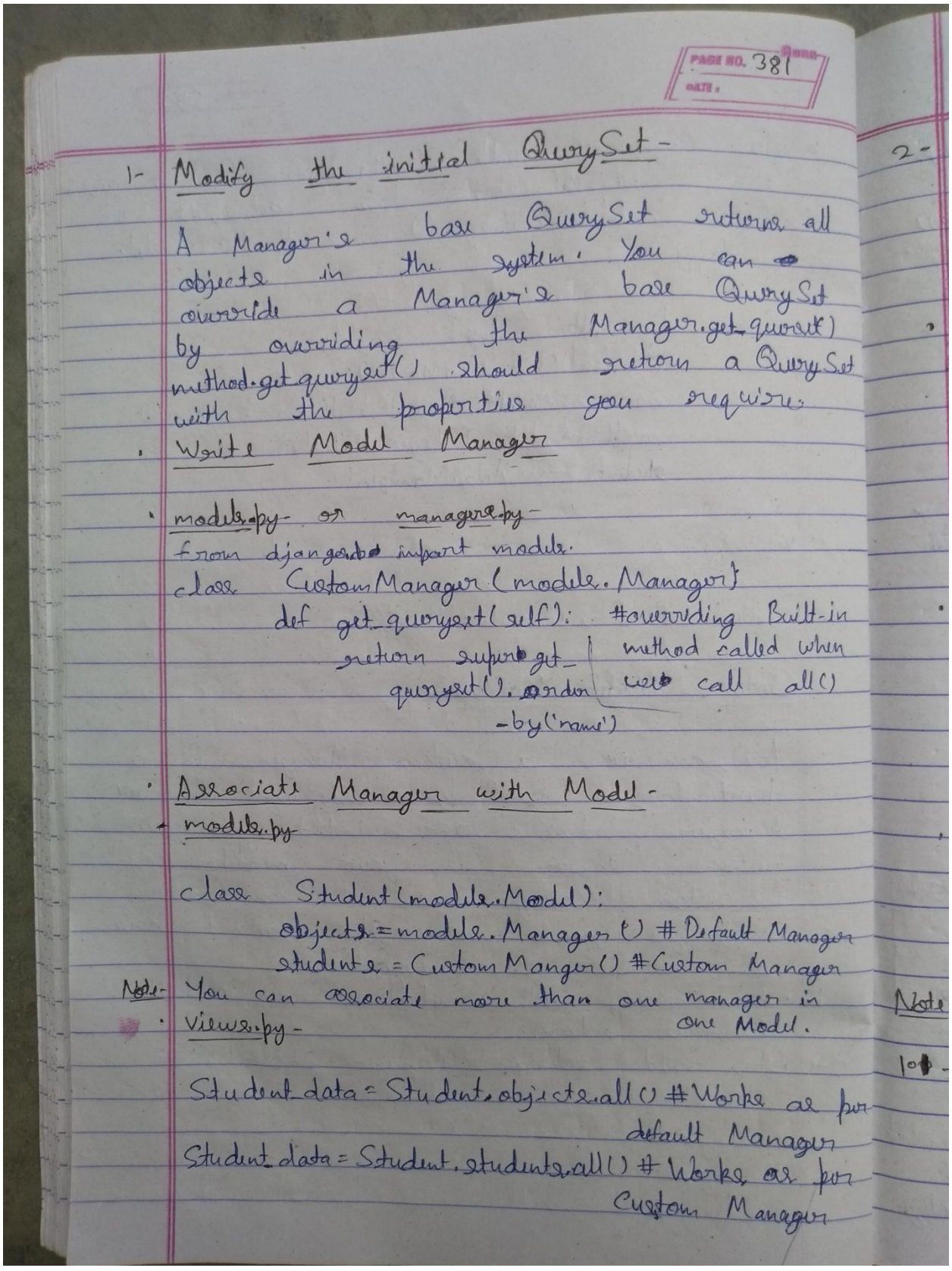
```
student_data = Student.students.all()
```

Custom Model Manager -

- You can use a custom Manager in a particular model by extending the base Manager class and instantiating your custom Manager in your model.
- A custom Manager method can return anything you want. It doesn't have to return a QuerySet.

1- • to modify the initial QuerySet the Manager returns.

2- • to add extra Manager methods.



PAGE NO. 381
DATE :

1- Modify the initial QuerySet -

A Manager's base QuerySet returns all objects in the system. You can override a Manager's base QuerySet by overriding the Manager.get_queryset() method. get_queryset() should return a QuerySet with the properties you require.

• Write Model Manager

• models.py - or managers.py -

```
from django import import models
class CustomManager(models.Manager):
    def get_queryset(self): #overriding Built-in method called when we call all()
        return super().get_queryset().order_by('name')
```

• Associate Manager with Model -

- models.py

```
class Student(models.Model):
    objects = models.Manager() # Default Manager
    students = CustomManger() # Custom Manager
```

Note - You can associate more than one manager in one Model.

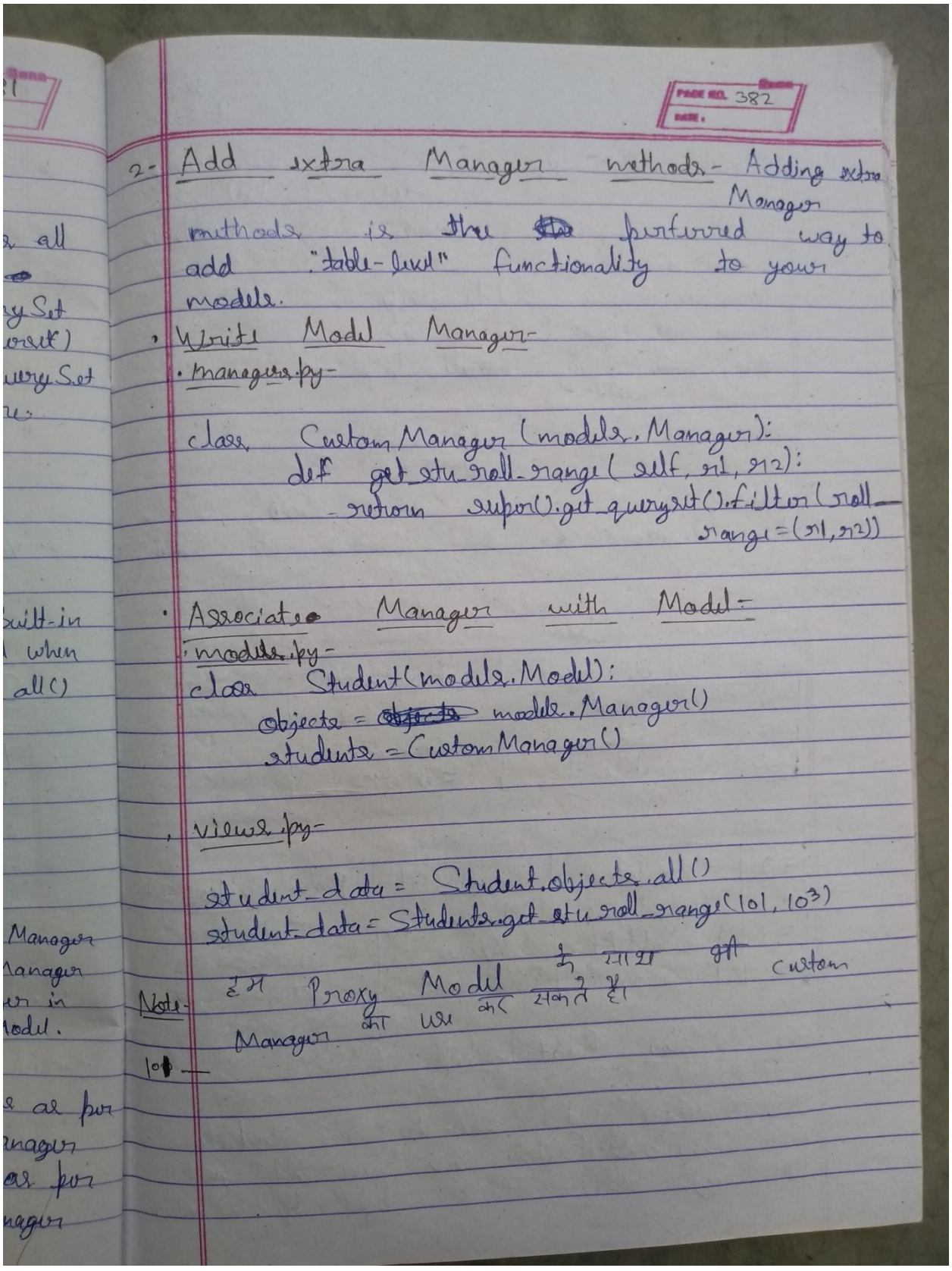
• Views.py -

```
Student_data = Student.objects.all() # Works as per default Manager
Student_data = Student.students.all() # Works as per Custom Manager
```

2-

Note

10



2- Add extra Manager methods - Adding extra Manager methods is the preferred way to add "table-level" functionality to your models.

• Write Modul Manager-

• managers.py-

```
class CustomManager(models.Manager):
    def get_stu_roll_range(self, r1, r2):
        return super().get_queryset().filter(roll__range=(r1, r2))
```

• Associate Manager with Modul-

• models.py-

```
class Student(models.Model):
    objects = models.Manager()
    students = CustomManager()
```

• views.py-

```
student_data = Student.objects.all()
student_data = Students.get_stu_roll_range(101, 103)
```

Note- हम Proxy Modul में भी custom Manager का use कर सकते है।

PAGE NO. 383
DATE : 11-8-20

Model Relationship - Django offers ways to define the 3 most common types of database relationships -

1- One to One Relationship
2- Many to One Relationship
3- Many to Many Relationship

1- One to One Relationship -

When one row of table A can be linked to one row of table B.

EX-

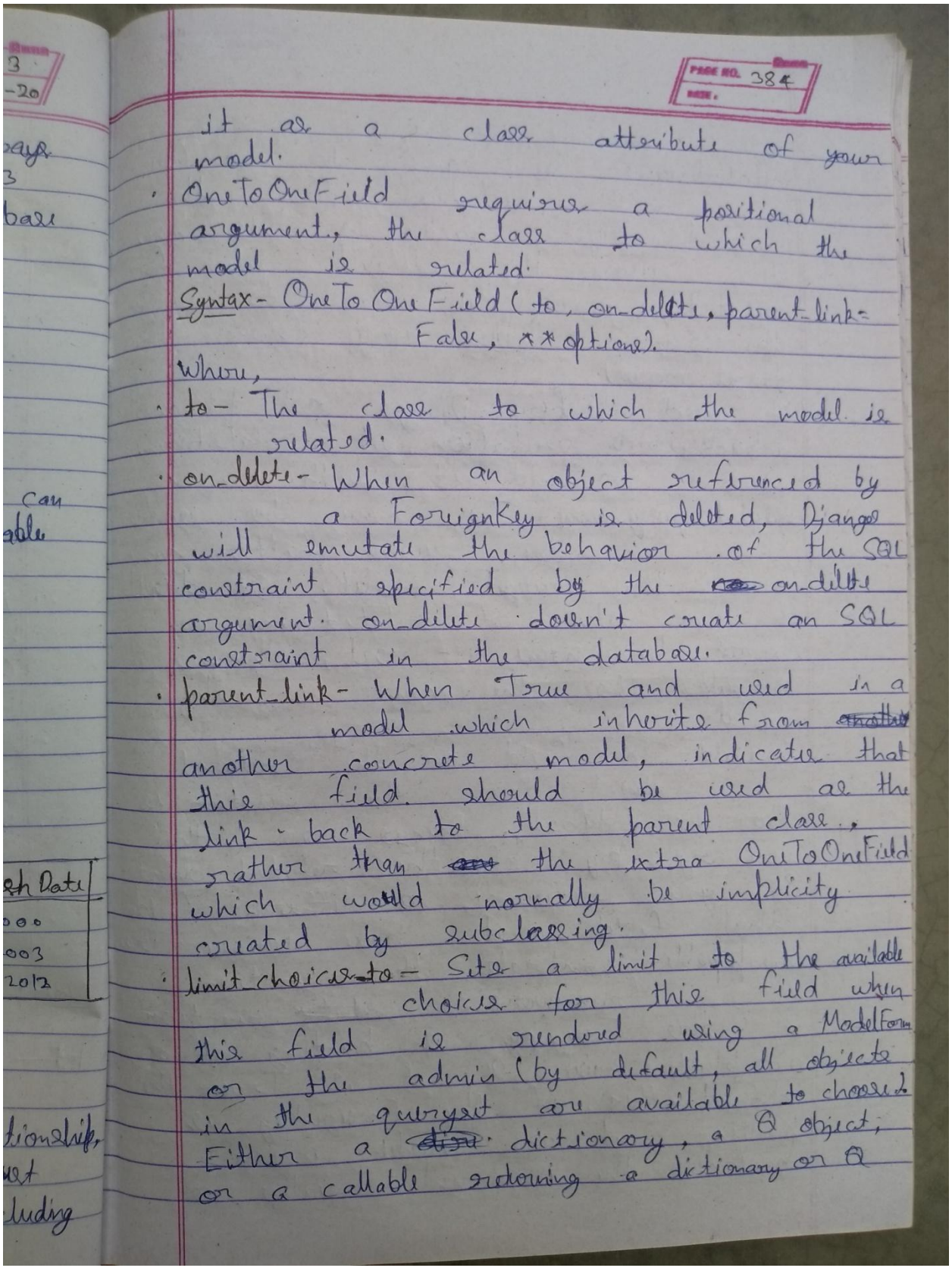
User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	Sonam12
3	Kunal	Kuhal12

ID	Page Name	Page Cat	Page Publish Date
1	Geekyshows	Programming	12-12-2000
2	World News	News	11-09-2003
3	Djangoone	Programming	07-01-2012

One to One Relationship -

To define a one-to-one relationship, use OneToOneField. You use it just like any other Field type by including



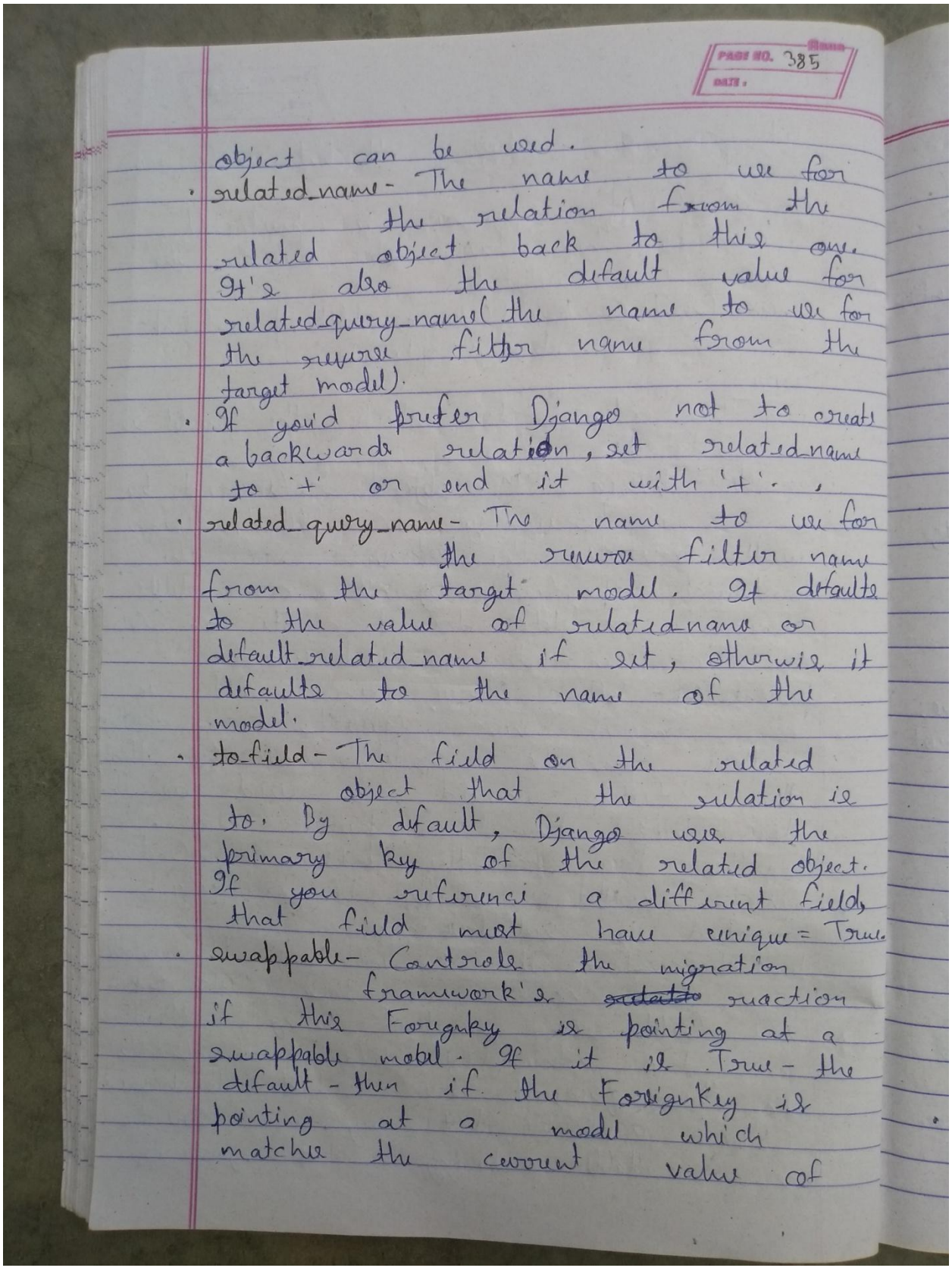
it as a class attribute of your model.

- OneToOneField requires a positional argument, the class to which the model is related.

Syntax - OneToOneField(to, on_delete, parent_link=False, **options).

Where,

- to - The class to which the model is related.
- on_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on_delete argument. on_delete doesn't create an SQL constraint in the database.
- parent_link - When True and used in a model which inherits from another concrete model, indicates that this field should be used as the link back to the parent class, rather than the extra OneToOneField which would normally be implicity created by subclassing.
- limit_choices_to - Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a Q object, or a callable returning a dictionary or Q



PAGE NO. 385
DATE :

object can be used.

- related_name - The name to use for the relation from the related object back to this one. It's also the default value for related_query_name (the name to use for the reverse filter name from the target model).
- If you'd prefer Django not to create a backwards relation, set related_name to '+' or end it with '+'.
- related_query_name - The name to use for the reverse filter name from the target model. It defaults to the value of related_name or default_related_name if set, otherwise it defaults to the name of the model.
- to_field - The field on the related object that the relation is to. By default, Django uses the primary key of the related object. If you reference a different field, that field must have unique = True.
- swappable - Controls the migration framework's reaction ~~related to~~ if this ForeignKey is pointing at a swappable model. If it is True - the default - then if the ForeignKey is pointing at a model which matches the current value of

setting.AUTH_USER_MODEL (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

- db_constraint - Control whether or not a constraint should be created in the database for this foreign key. The default is True. and that's almost certainly what you want; setting this to False can be very bad for data integrity. That said, here are some scenarios where you might want to do this -
  - You have legacy data that is not valid.
  - You're sharding you database.
  - If this is set to False, accessing a related object that doesn't exist will raise its DoesNotExist exception.

- on_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint by the on_delete argument. on_delete doesn't create an SQL constraint in the database.

- The possible values for on_delete are found in django.db.models -

PAGE NO. 387
DATE :

(i) CASCADE - Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ~~Foreign~~ ForeignKey.

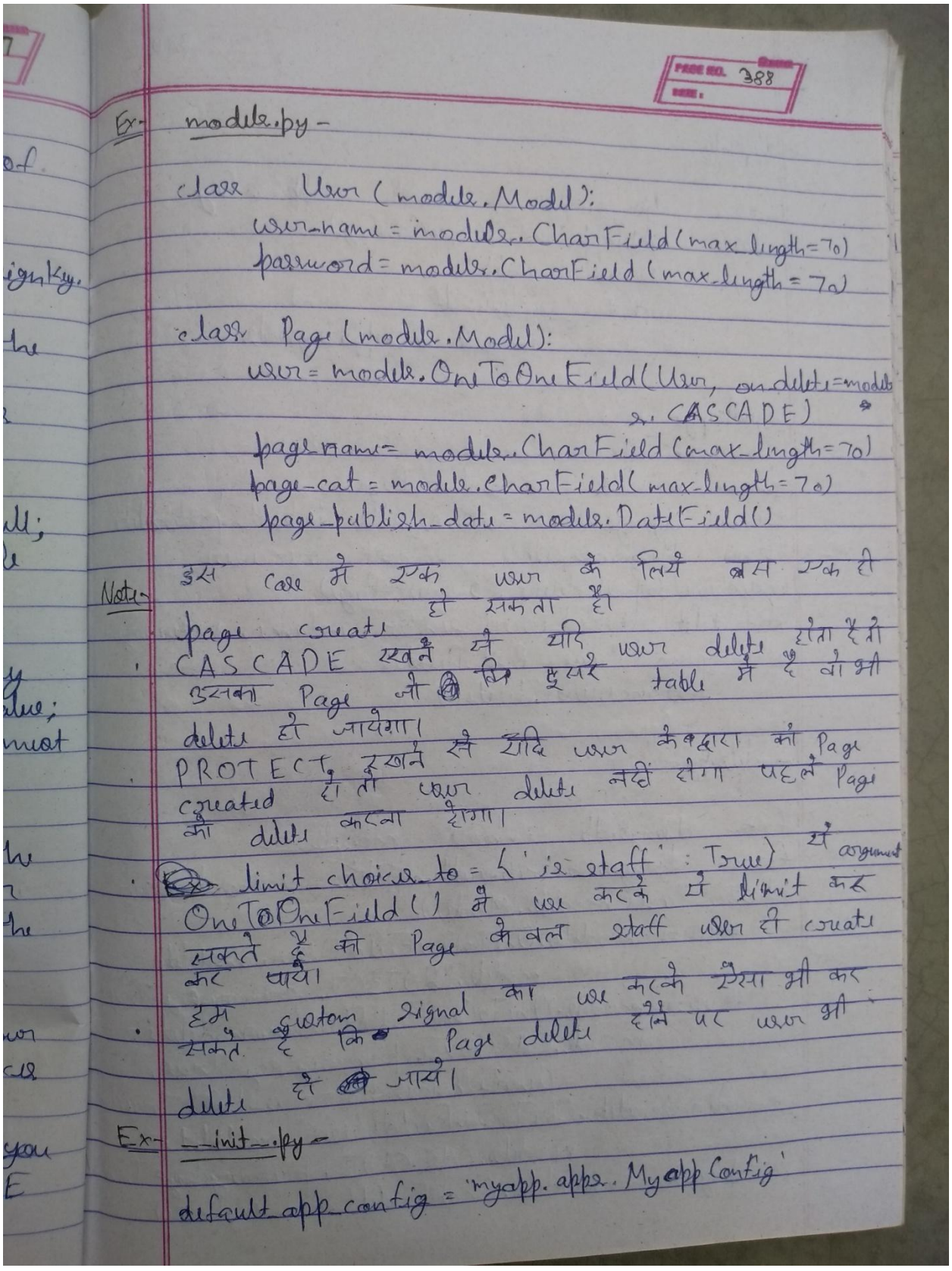
(ii) PROTECT - Prevent deletion of the referenced object by raising ProtectedError, a subclass of django.db.IntegrityError.

(iii) SET_NULL - Set the ForeignKey null; this is only possible if ~~null~~ null is True.

(iv) SET_DEFAULT - Set the ForeignKey to its default value; a default for the ForeignKey must be set.

(v) SET() - Set the ForeignKey to the value passed to SET(), or if a callable is passed in, the result of calling it.

(vi) DO_NOTHING - Take no action. If your database backend enforces referential integrity, this will cause an IntegrityError unless you manually add an SQL ON DELETE constraint to the database field.



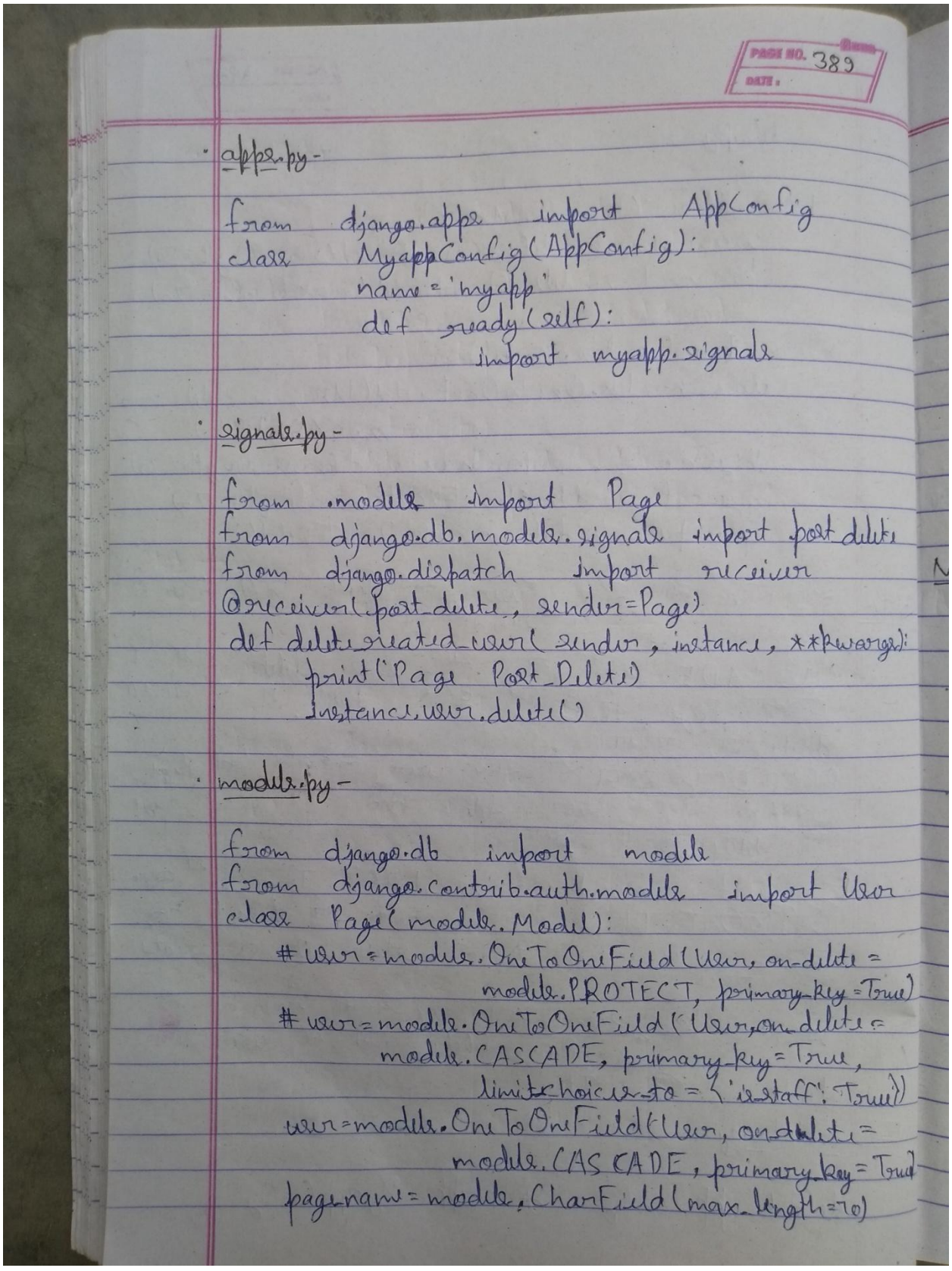
Ex- module.py -

```
class User(models.Model):
    user_name = models.CharField(max_length=70)
    password = models.CharField(max_length=70)

class Page(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    page_name = models.CharField(max_length=70)
    page_cat = models.CharField(max_length=70)
    page_publish_date = models.DateField()
```

Note-
- इस case में एक user के लिये बस एक ही page create हो सकता है
- CASCADE रखने से यदि user delete होता है तो उसका Page तो किसी दूसरे table में है वो भी delete हो जायेगा।
- PROTECT रखने से यदि user के द्वारा कोई Page created है तो user delete नहीं होगा पहले Page को delete करना होगा।
- limit_choices_to = {'is_staff': True} यह argument OneToOneField() में use करके हम limit कर सकते हैं की Page के वल staff user ही create कर पाये।
- हम custom signal का use करके ऐसा भी कर सकते हैं कि Page delete होने पर user भी delete हो जायें।

Ex- __init__.py -

```
default_app_config = 'myapp.apps.MyappConfig'
```



PAGE NO. 389
DATE :

- apps.py -

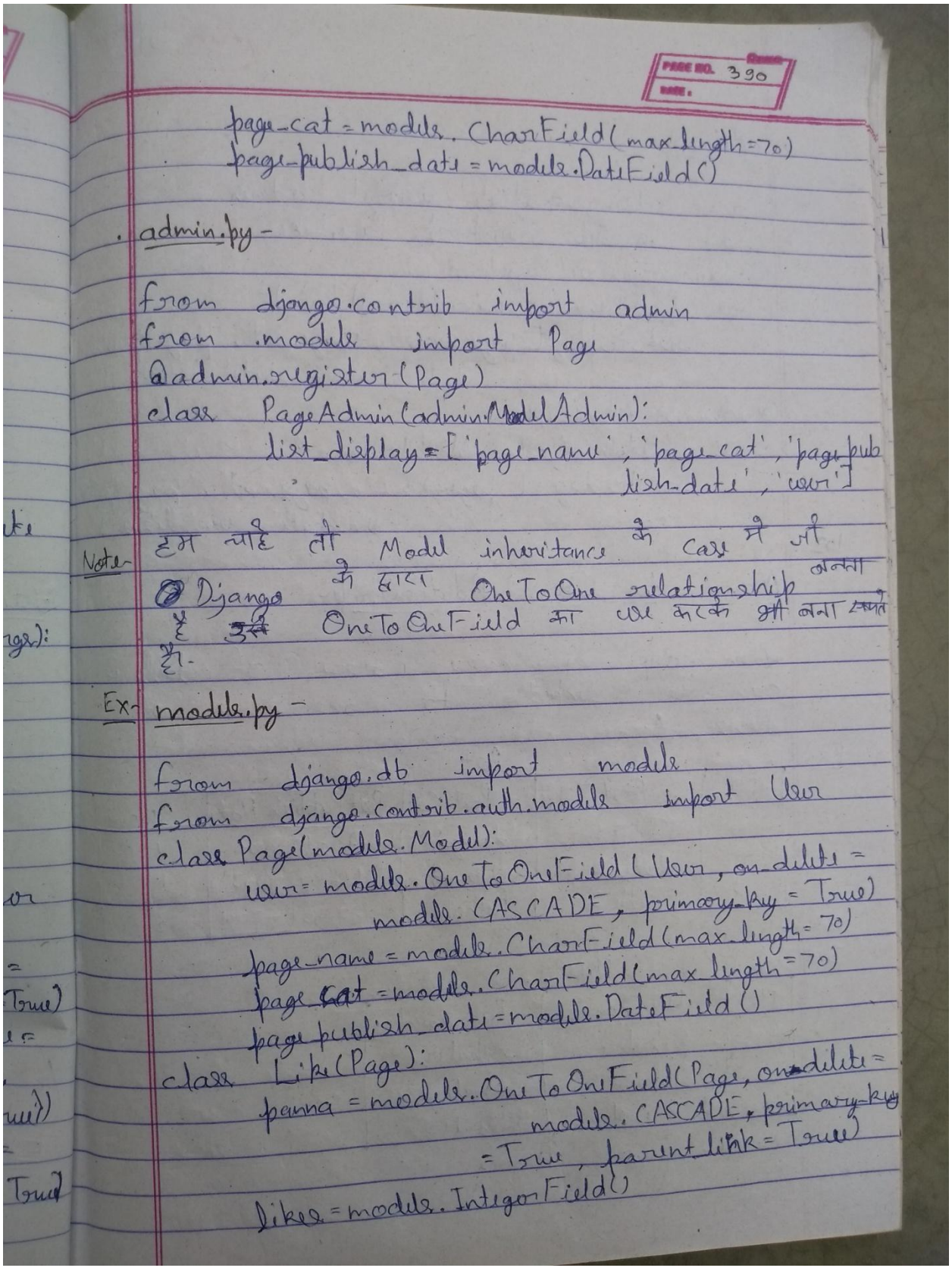
```
from django.apps import AppConfig
class MyappConfig(AppConfig):
    name = 'myapp'
    def ready(self):
        import myapp.signals
```

- signals.py -

```
from .models import Page
from django.db.models.signals import post_delete
from django.dispatch import receiver
@receiver(post_delete, sender=Page)
def delete_related_user(sender, instance, **kwargs):
    print('Page Post_Delete')
    instance.user.delete()
```

- models.py -

```
from django.db import models
from django.contrib.auth.models import User
class Page(models.Model):
    # user = models.OneToOneField(User, on_delete =
            models.PROTECT, primary_key = True)
    # user = models.OneToOneField(User, on_delete =
            models.CASCADE, primary_key = True,
            limit_choices_to = {'is_staff': True})
    user = models.OneToOneField(User, on_delete =
            models.CASCADE, primary_key = True)
    page_name = models.CharField(max_length=70)
```



page_cat = models.CharField(max_length=70)
page_publish_date = models.DateField()

. admin.py -

```
from django.contrib import admin
from .models import Page
@admin.register(Page)
class PageAdmin(admin.ModelAdmin):
    list_display = ['page_name', 'page_cat', 'page_publish_date', 'user']
```

Note - हम चाहे तो Model inheritance के case में भी Django के द्वारा OneToOne relationship बना ~~सकते~~ है ~~है~~ OneToOneField का use करके भी बना सकते है.

Ex- models.py -

```
from django.db import models
from django.contrib.auth.models import User
class Page(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, primary_key=True)
    page_name = models.CharField(max_length=70)
    page_cat = models.CharField(max_length=70)
    page_publish_date = models.DateField()
class Link(Page):
    panna = models.OneToOneField(Page, on_delete=models.CASCADE, primary_key=True, parent_link=True)
    likes = models.IntegerField()
```

PAGE NO. 391
DATE :

admin.py-

```
from django.contrib import admin
from .models import Page, Like
@admin.register(Page)
class PageAdmin(admin.ModelAdmin):
    list_display=['page_name', 'page_cat', 'page_publish_date', 'user']

@admin.register(Like)
class LikeAdmin(admin.ModelAdmin):
    list_display = ['page_name', 'page_cat', 'page_publish_date', 'user', 'like']
```

102 —

2- Many to One Relationship-

When one or more row of table B can be linked to one row of table A.

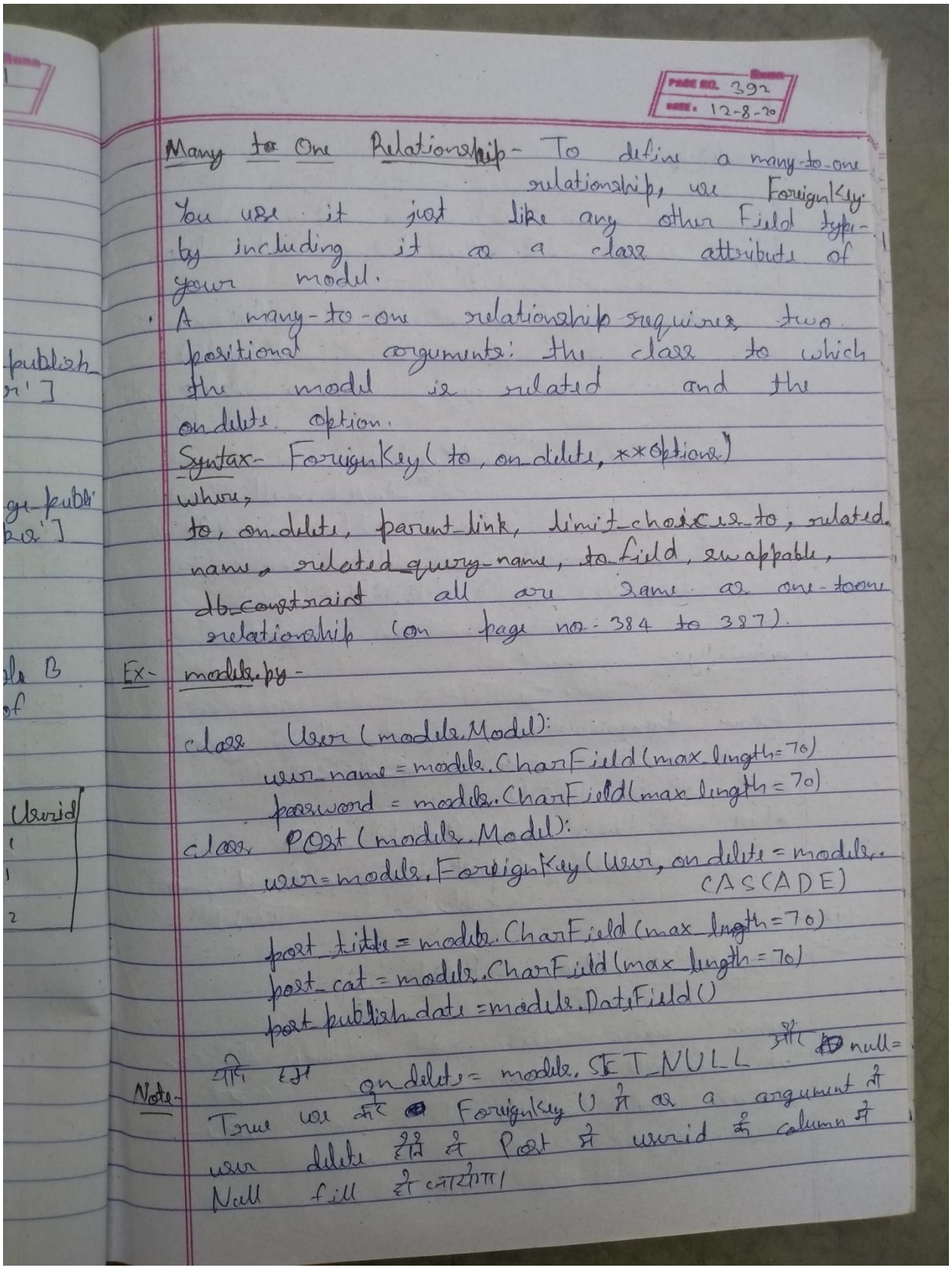
Ex-

Post

ID	Post Title	Post Cat	Post Publish Date	Userid
1	Title 1	django	12-12-200	1
2	Title 2	django	11-09-2003	1
3	Title 3	python	07-01-2001	2

User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	sonal12
3	Kunal	Kunal12



PAGE NO. 392
DATE: 12-8-20

Many to One Relationship - To define a many-to-one relationship, use ForeignKey. You use it just like any other Field type - by including it as a class attribute of your model.

A many-to-one relationship requires two positional arguments: the class to which the model is related and the on_delete option.

Syntax - ForeignKey(to, on_delete, **options)

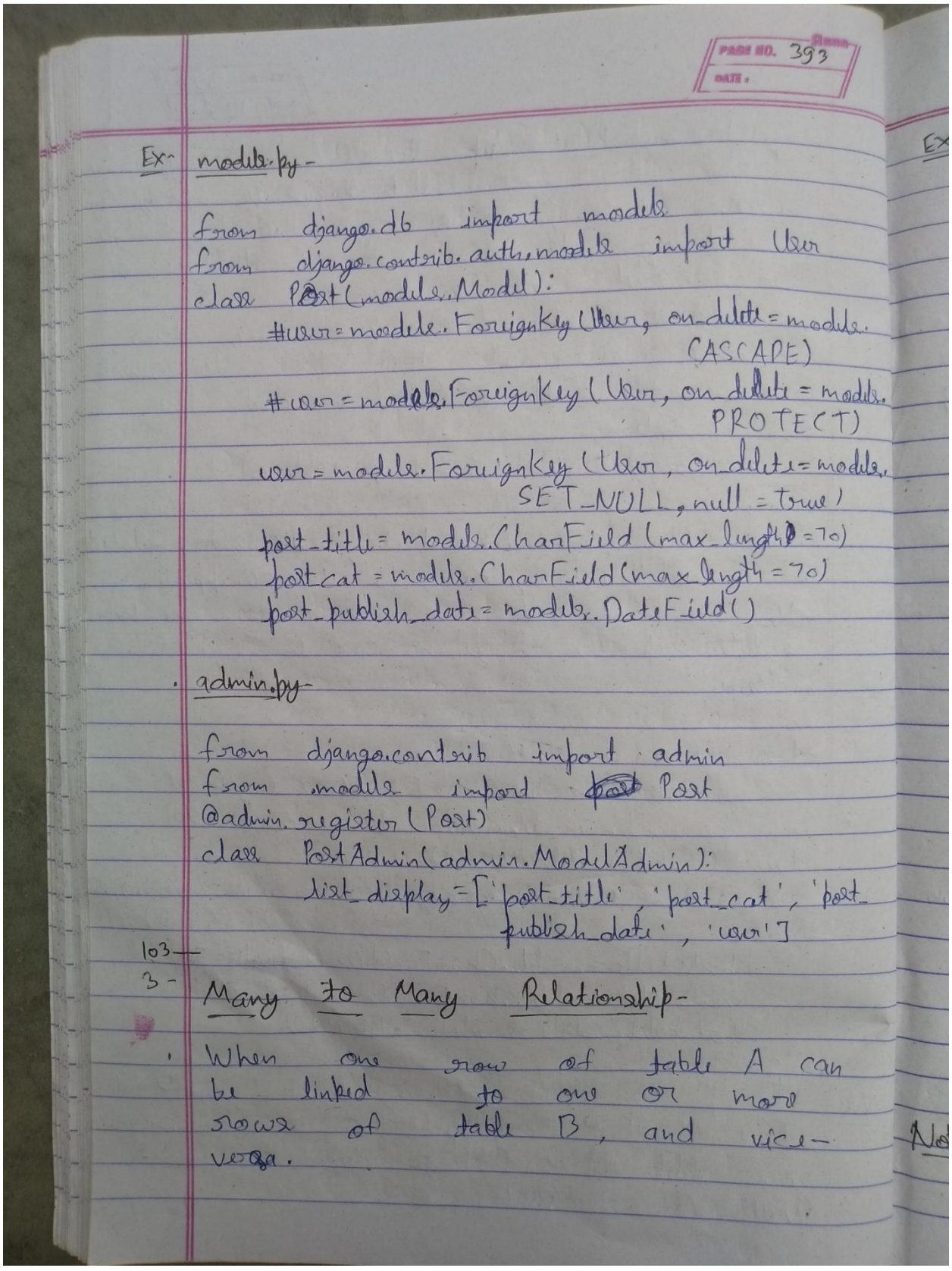
where,

to, on_delete, parent_link, limit_choices_to, related_name, related_query_name, to_field, swappable, db_constraint all are same as one-to-one relationship (on page no - 384 to 387).

Ex - models.py -

```
class User(models.Model):
    user_name = models.CharField(max_length=70)
    password = models.CharField(max_length=70)
class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    post_title = models.CharField(max_length=70)
    post_cat = models.CharField(max_length=70)
    post_publish_date = models.DateField()
```

Note - यदि हम on_delete = models.SET_NULL यदि null = True का use करें Foreignkey () में as a argument में user delete होने से Post में userid के column में Null fill हो जायेगा।



PAGE NO. 393
DATE :

Ex^ models.py -

```
from django.db import models
from django.contrib.auth.models import User
class Post(models.Model):
    #user = models.ForeignKey(User, on_delete = models.
                                        CASCADE)
    #user = models.ForeignKey(User, on_delete = models.
                                        PROTECT)
    user = models.ForeignKey(User, on_delete = models.
                                SET_NULL, null = true)
    post_title = models.CharField(max_length = 70)
    post_cat = models.CharField(max_length = 70)
    post_publish_date = models.DateField()
```

. admin.py -

```
from django.contrib import admin
from models import Post
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['post_title', 'post_cat', 'post_
                    publish_date', 'user']
```

103 —

3 - Many to Many Relationship -

. When one row of table A can be linked to one or more rows of table B, and vice-versa.

Ex-

User

ID	Username	Password
1	Rahul	Rahul12
2	Sonam	Sonam12
3	Kunal	Kunal12

Song

ID	Song Name	Song Duration
1	Tum hi ho	5
2	Kuch Kuch	7
3	Dil to hai dil	8

Song_user

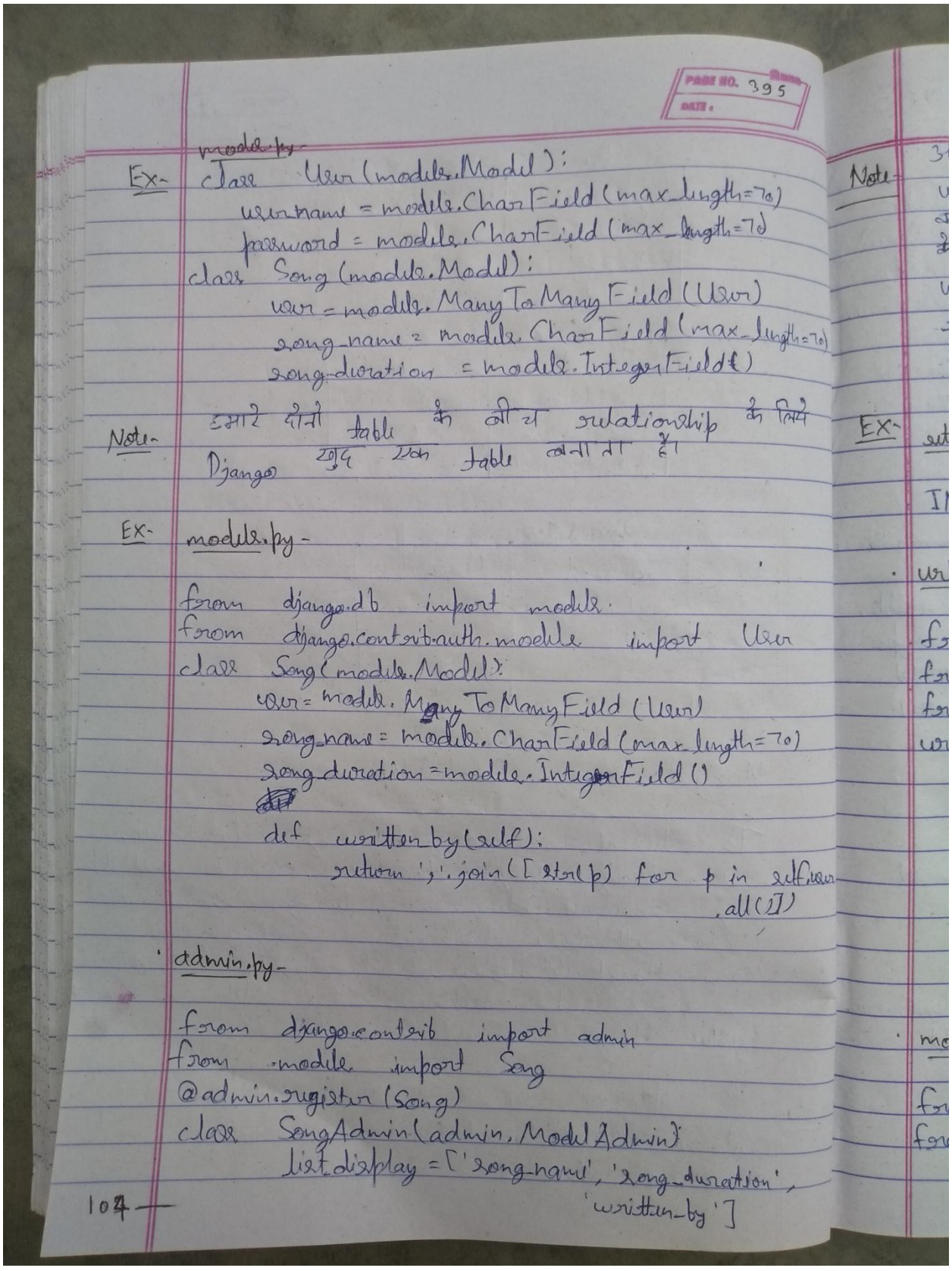
ID	Song_id	User_id
1	1	1
2	1	2
3	2	3
4	2	1

Many to Many Relationship-

- To define a many-to-many relationship, use ManyToManyField. You use it just like any other Field - by including it as a class attribute of your model.
- ManyToManyField requires a positional argument - the class to which the model is related.

Syntax- ManyToManyField(to, **options)

Note- बाकी options like to, on_delete etc. same ही है OneToOne और Many to One relationship की तरह पर कुछ-कुछ अलग भी है।



PAGE NO. 395
DATE :

models.py -

Ex- class User(models.Model):
    username = models.CharField(max_length=70)
    password = models.CharField(max_length=70)
class Song(models.Model):
    user = models.ManyToManyField(User)
    song_name = models.CharField(max_length=70)
    song_duration = models.IntegerField()

Note- हमारे दोनों table के बीच relationship के लिये Django एक नया table बनाता है।

Ex- models.py -

from django.db import models
from django.contrib.auth.models import User
class Song(models.Model):
    user = models.ManyToManyField(User)
    song_name = models.CharField(max_length=70)
    song_duration = models.IntegerField()
    def written_by(self):
        return ','.join([str(p) for p in self.user.all()])

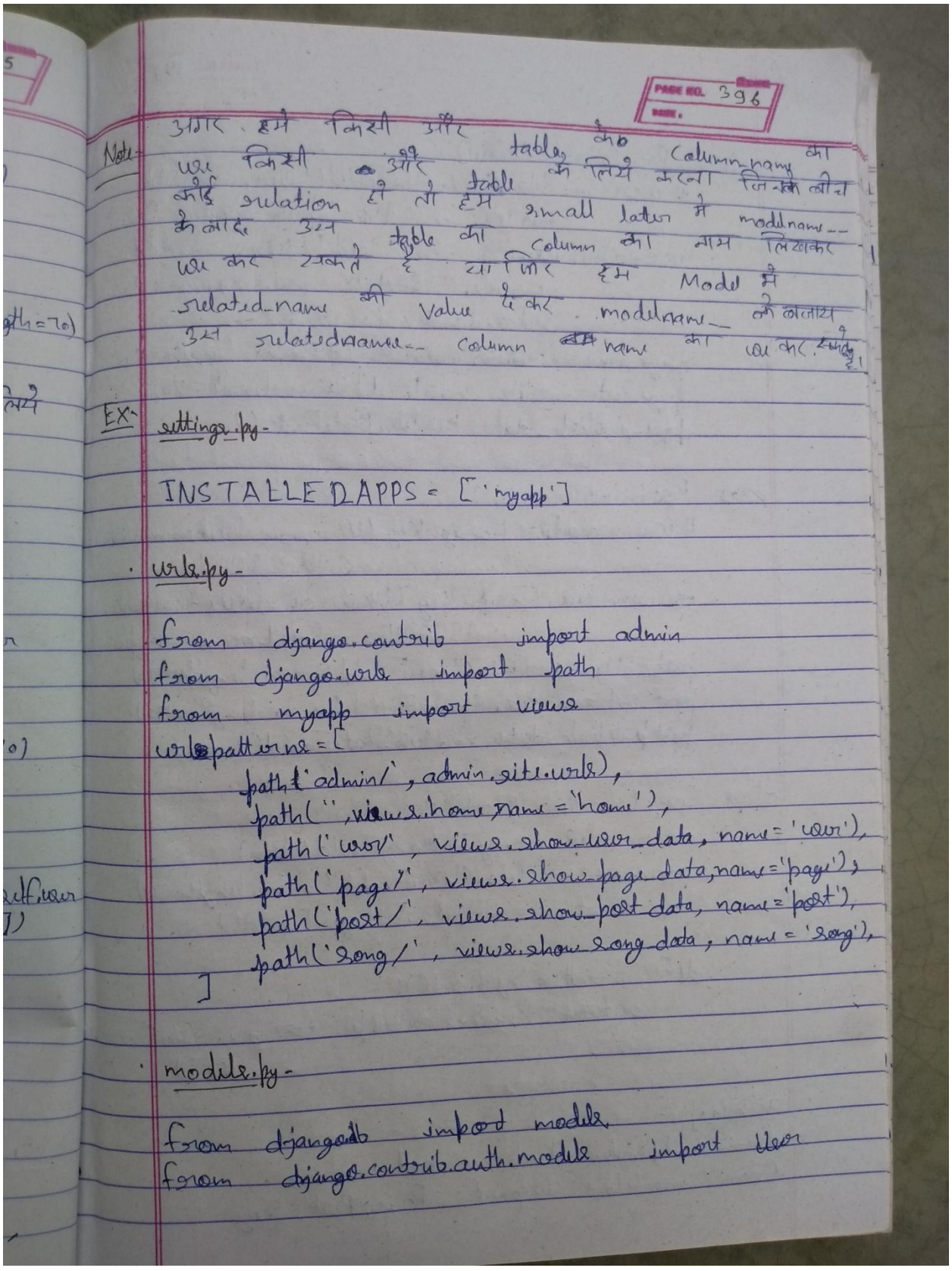
admin.py -

from django.contrib import admin
from .models import Song
@admin.register(Song)
class SongAdmin(admin.ModelAdmin):
    list_display = ['song_name', 'song_duration', 'written_by']

Note-

Ex-

104



Note:- अगर हमें किसी और table के column_name का use किसी और table के लिये करना है तो दोनों के बीच कोई relation है तो हम small letter में modulname__ के बाद उस table का column का नाम लिखकर use कर सकते है या फिर हम Model में related_name की value देकर modulname__ के बाद उस related_name__ column name का use कर सकते है।

EX:- settings.py-

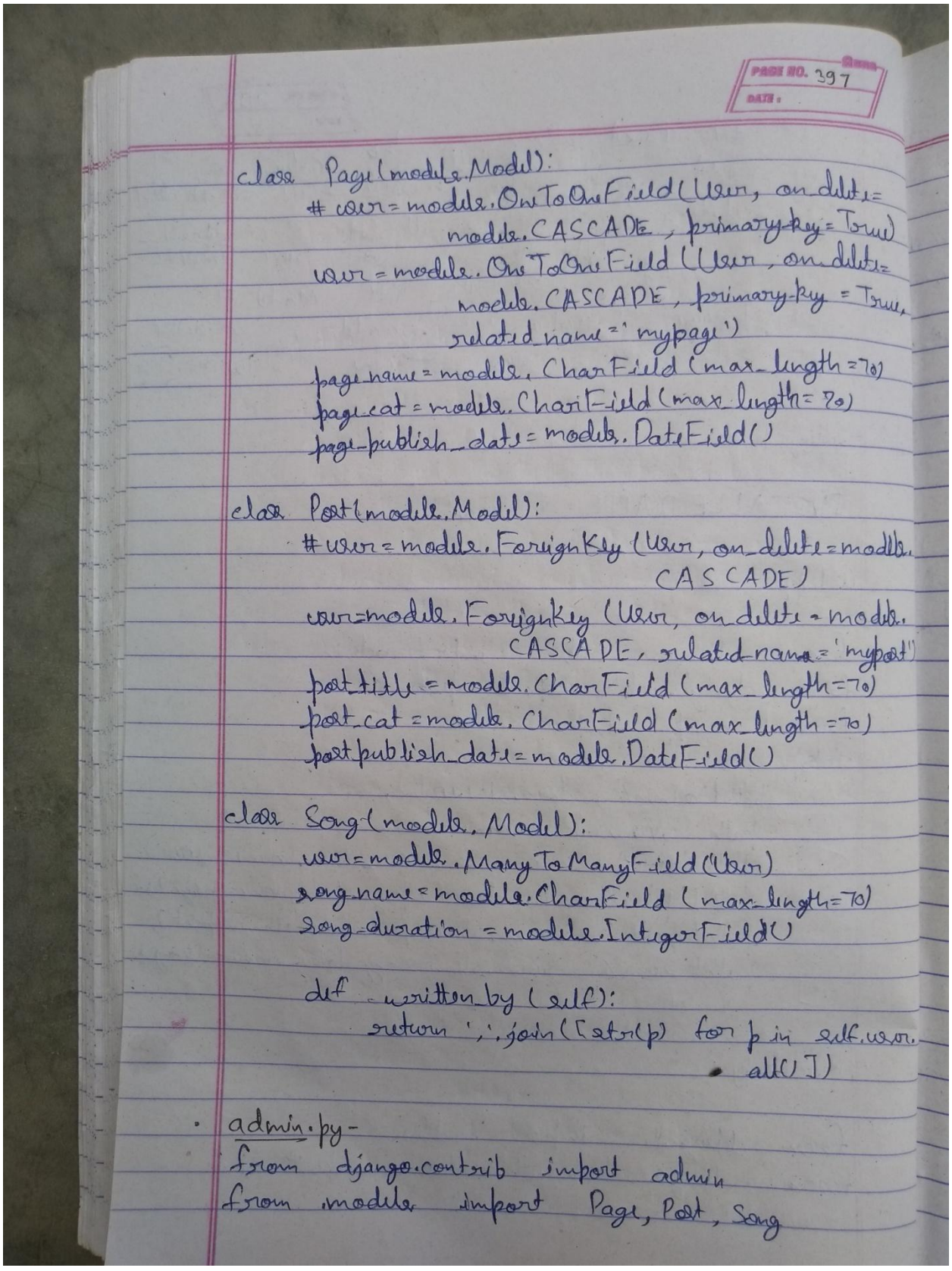
```
INSTALLED_APPS = ['myapp']
```

- urls.py-

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home, name='home'),
    path('user', views.show_user_data, name='user'),
    path('page/', views.show_page_data, name='page'),
    path('post/', views.show_post_data, name='post'),
    path('song/', views.show_song_data, name='song'),
]
```

- models.py-

```
from django.db import models
from django.contrib.auth.models import User
```

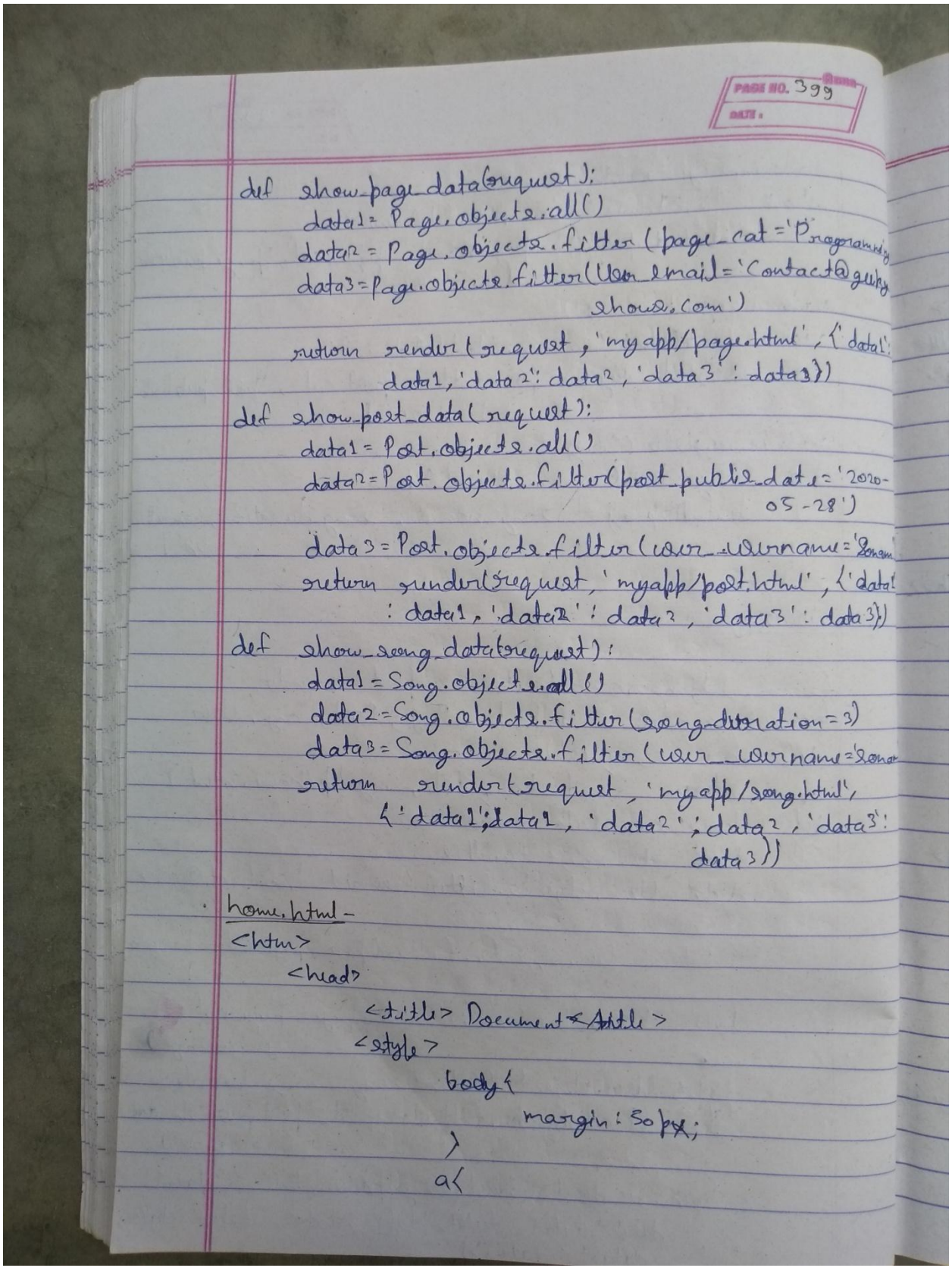


PAGE NO. 397
DATE :

```
class Page(models.Model):
    #user = models.OneToOneField(User, on_delete=
                models.CASCADE, primary_key=True)
    user = models.OneToOneField(User, on_delete=
                models.CASCADE, primary_key = True,
                related_name='mypage')
    page_name = models.CharField(max_length=70)
    page_cat = models.CharField(max_length=70)
    page_publish_date = models.DateField()

class Post(models.Model):
    #user = models.ForeignKey(User, on_delete=models.
                                    CASCADE)
    user=models.ForignKey(User, on_delete=models.
                CASCADE, related_name='mypost')
    post_title = models.CharField(max_length=70)
    post_cat = models.CharField(max_length=70)
    post_publish_date=models.DateField()

class Song(models.Model):
    user=models.ManyToManyField(User)
    song_name=models.CharField(max_length=70)
    song_duration = models.IntigerField()

    def written_by(self):
        return ','.join([str(p) for p in self.user.
                                    all()])
```

- admin.py -

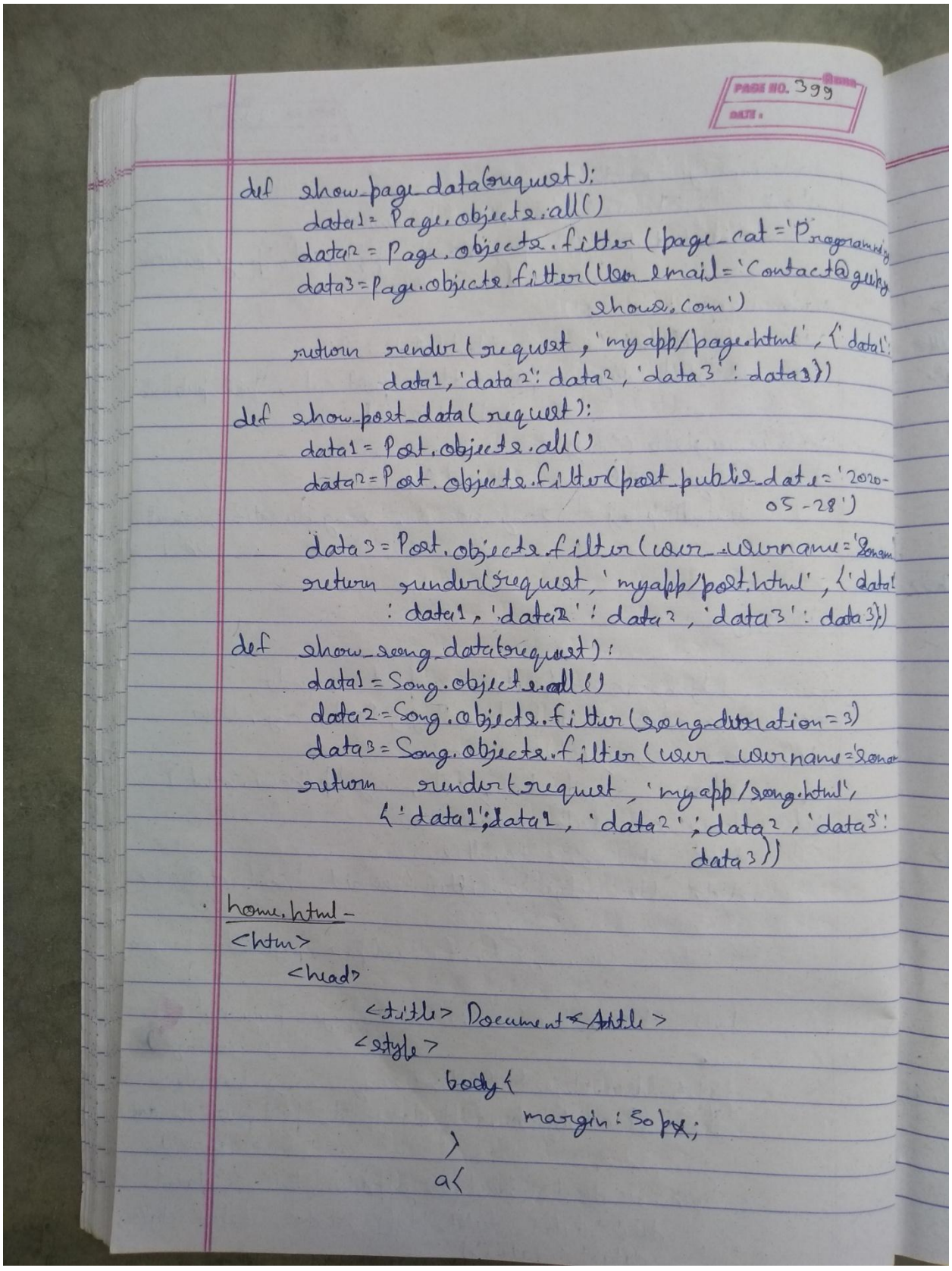
```
from django.contrib import admin
from models import Page, Post, Song
```



```
def show_page_data(request):
    data1 = Page.objects.all()
    data2 = Page.objects.filter(page_cat='Programming')
    data3 = Page.objects.filter(User_email='contact@geekyshows.com')
    return render(request, 'myapp/page.html', {'data1': data1, 'data2': data2, 'data3': data3})
def show_post_data(request):
    data1 = Post.objects.all()
    data2 = Post.objects.filter(post_publish_date='2020-05-28')
    data3 = Post.objects.filter(user_username='Sonam')
    return render(request, 'myapp/post.html', {'data1': data1, 'data2': data2, 'data3': data3})
def show_song_data(request):
    data1 = Song.objects.all()
    data2 = Song.objects.filter(song_duration=3)
    data3 = Song.objects.filter(user_username='Sonam')
    return render(request, 'myapp/song.html', {'data1': data1, 'data2': data2, 'data3': data3})
```

home.html -

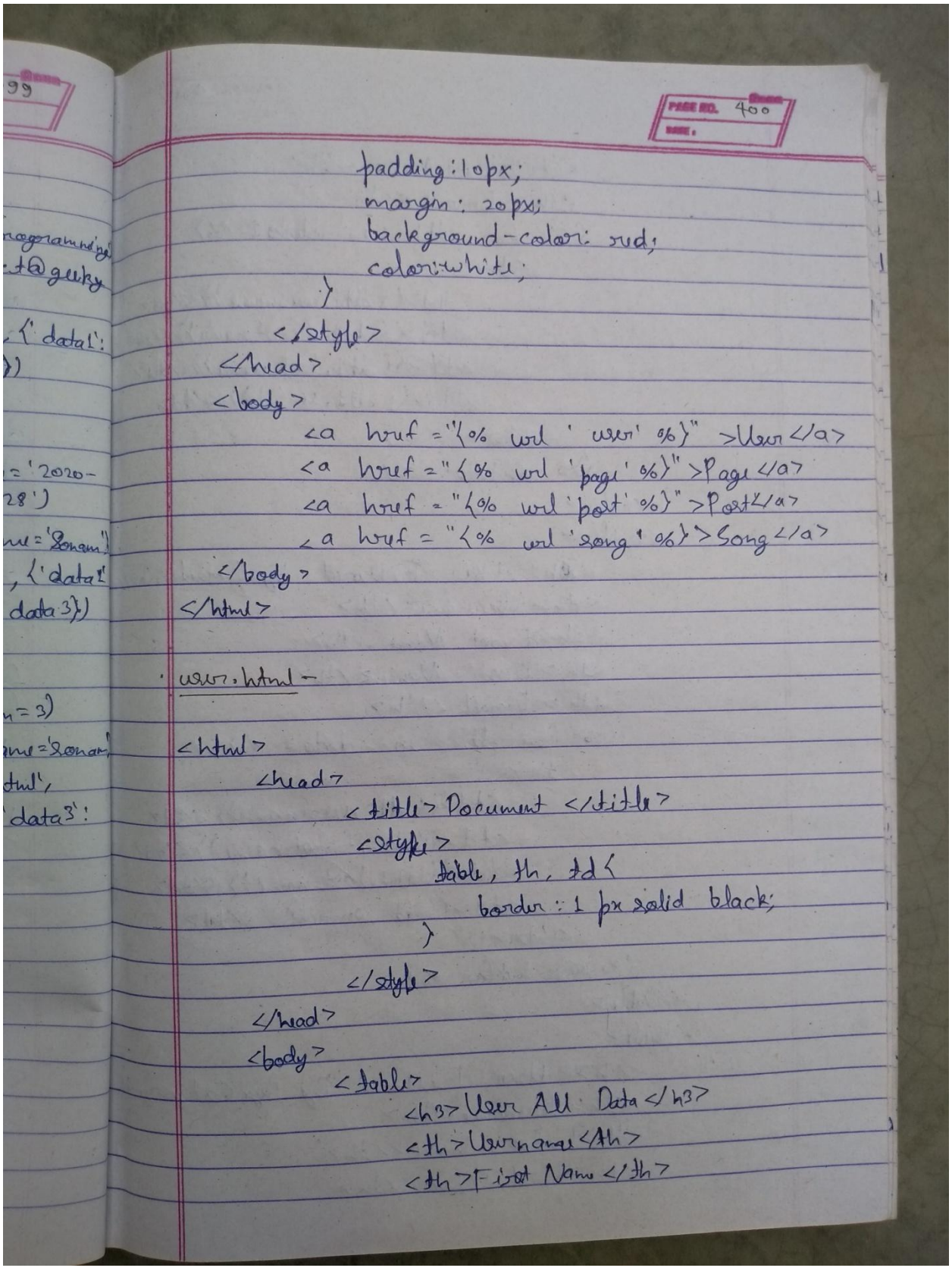
```
<html>
    <head>
        <title> Document </title>
        <style>
            body{
                margin: 50px;
            }
            a{
```



```
def show_page_data(request):
    data1 = Page.objects.all()
    data2 = Page.objects.filter(page_cat='Programming')
    data3 = Page.objects.filter(User_email='contact@gmail.com')
    return render(request, 'myapp/page.html', {'data1': data1, 'data2': data2, 'data3': data3})
def show_post_data(request):
    data1 = Post.objects.all()
    data2 = Post.objects.filter(post_publish_date='2020-05-28')
    data3 = Post.objects.filter(user_username='Sonam')
    return render(request, 'myapp/post.html', {'data1': data1, 'data2': data2, 'data3': data3})
def show_song_data(request):
    data1 = Song.objects.all()
    data2 = Song.objects.filter(song_duration=3)
    data3 = Song.objects.filter(user_username='Sonam')
    return render(request, 'myapp/song.html', {'data1': data1, 'data2': data2, 'data3': data3})
```

home.html -

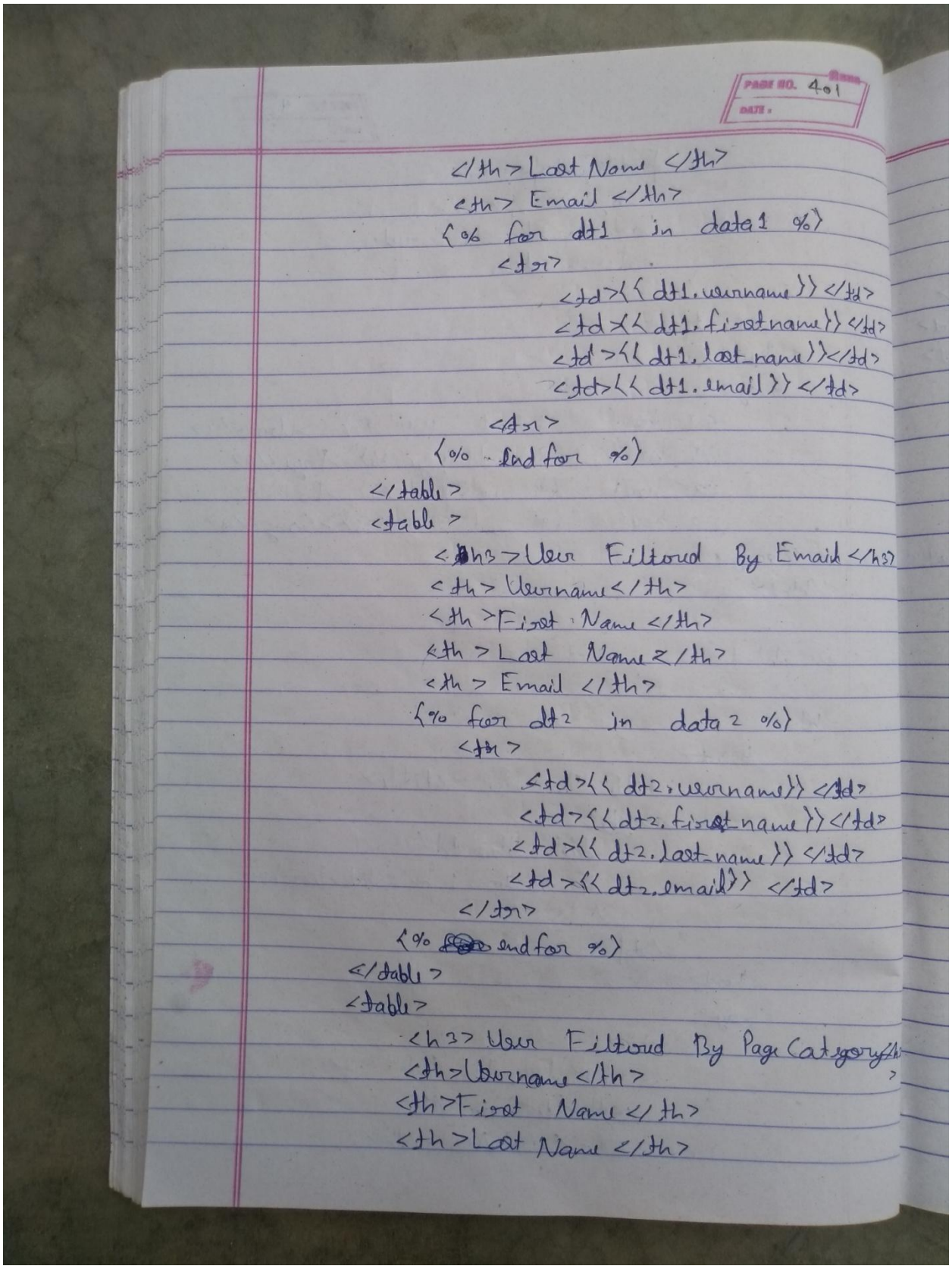
```
<html>
    <head>
        <title> Document </title>
        <style>
            body{
                margin: 50px;
            }
            a{
```



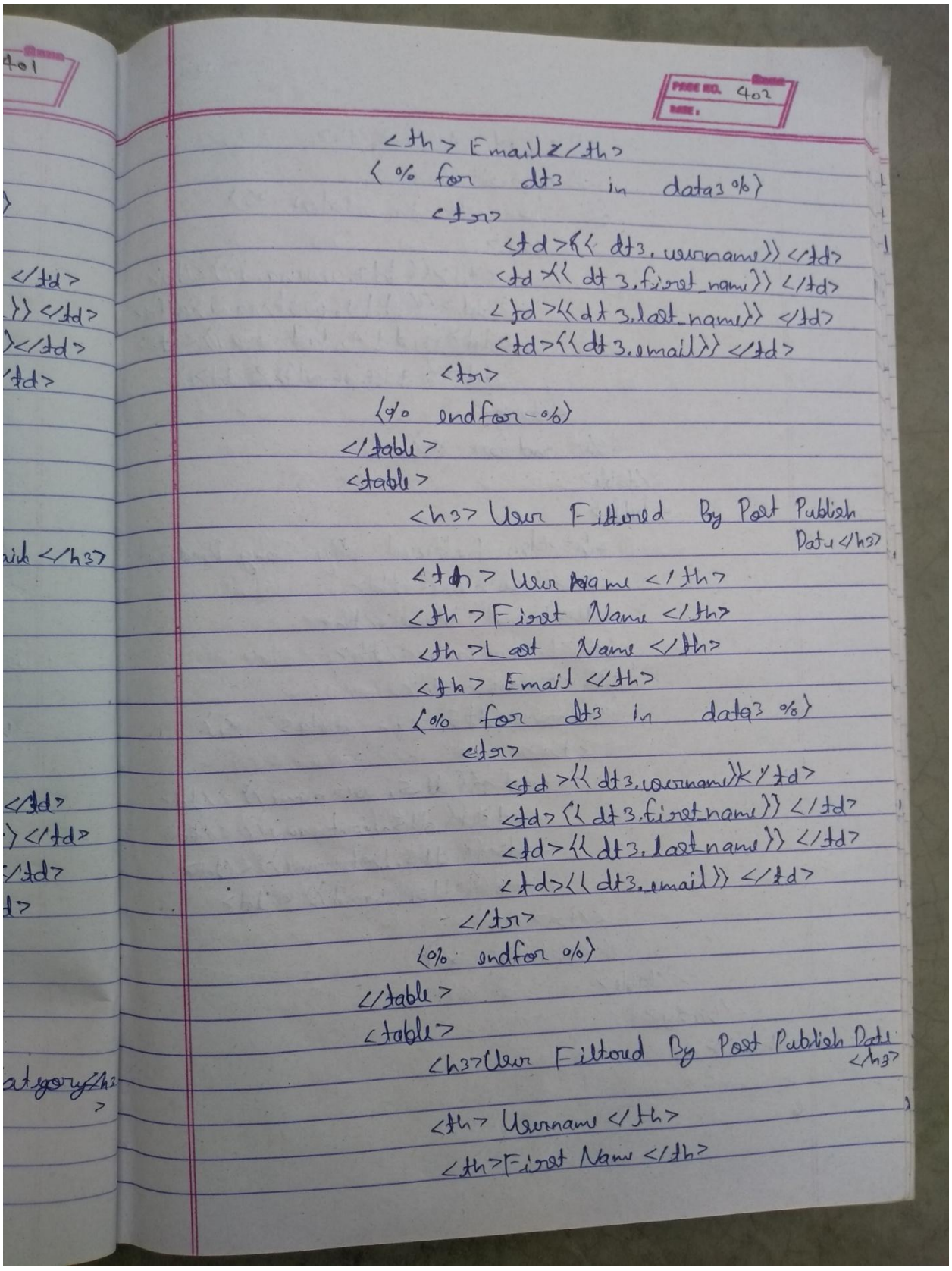
```
            padding:10px;
            margin: 20px;
            background-color: red;
            color:white;
        }
    </style>
</head>
<body>
    <a href="{% url 'user' %}" >User</a>
    <a href="{% url 'page' %}" >Page</a>
    <a href="{% url 'post' %}" >Post</a>
    <a href="{% url 'song' %}" >Song</a>
</body>
</html>
```

user.html -

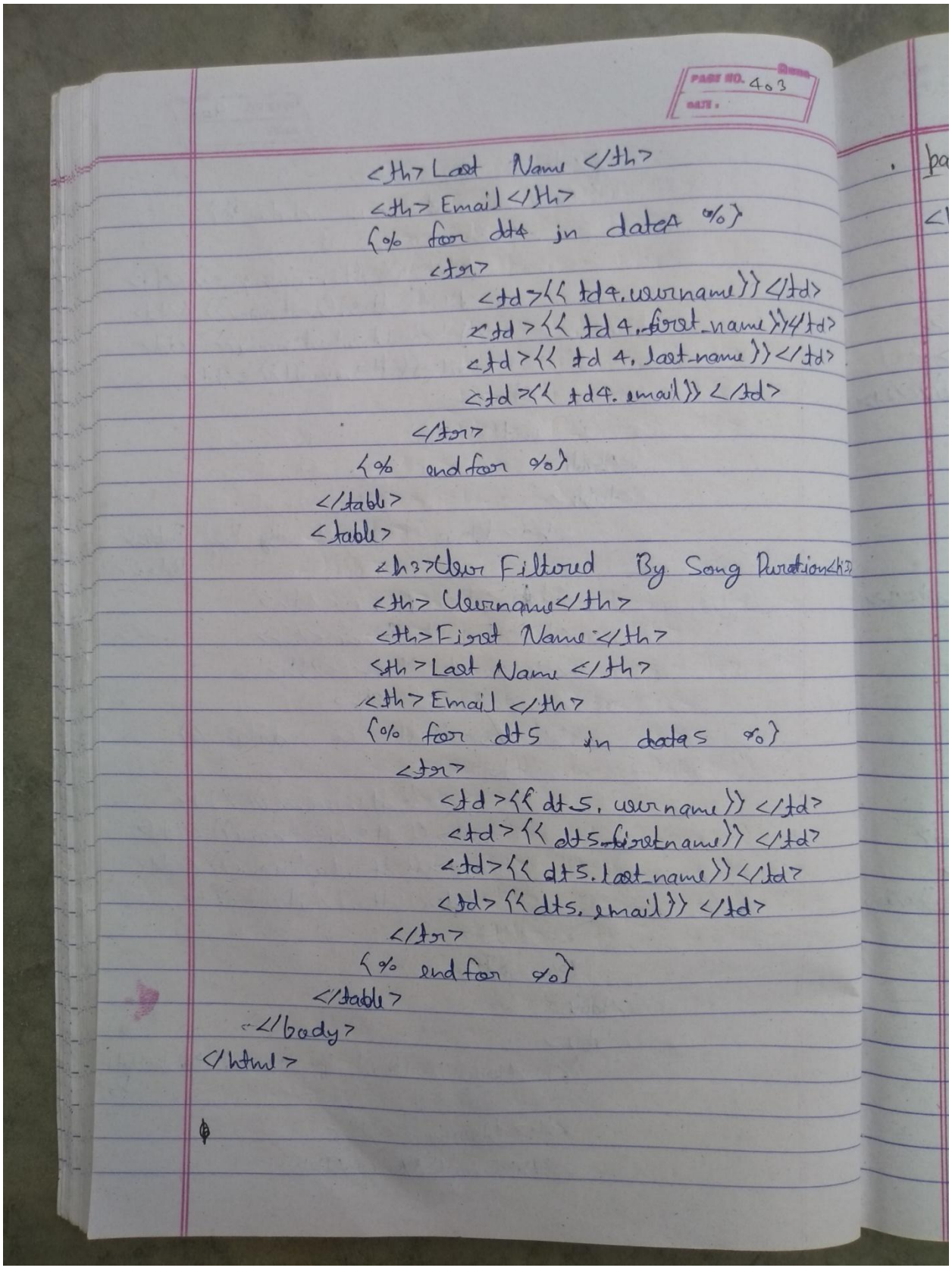
```
<html>
    <head>
        <title> Document </title>
        <style>
            table, th, td {
                border: 1px solid black;
            }
        </style>
    </head>
    <body>
        <table>
            <h3> User All Data </h3>
            <th> Username </th>
            <th> First Name </th>
```



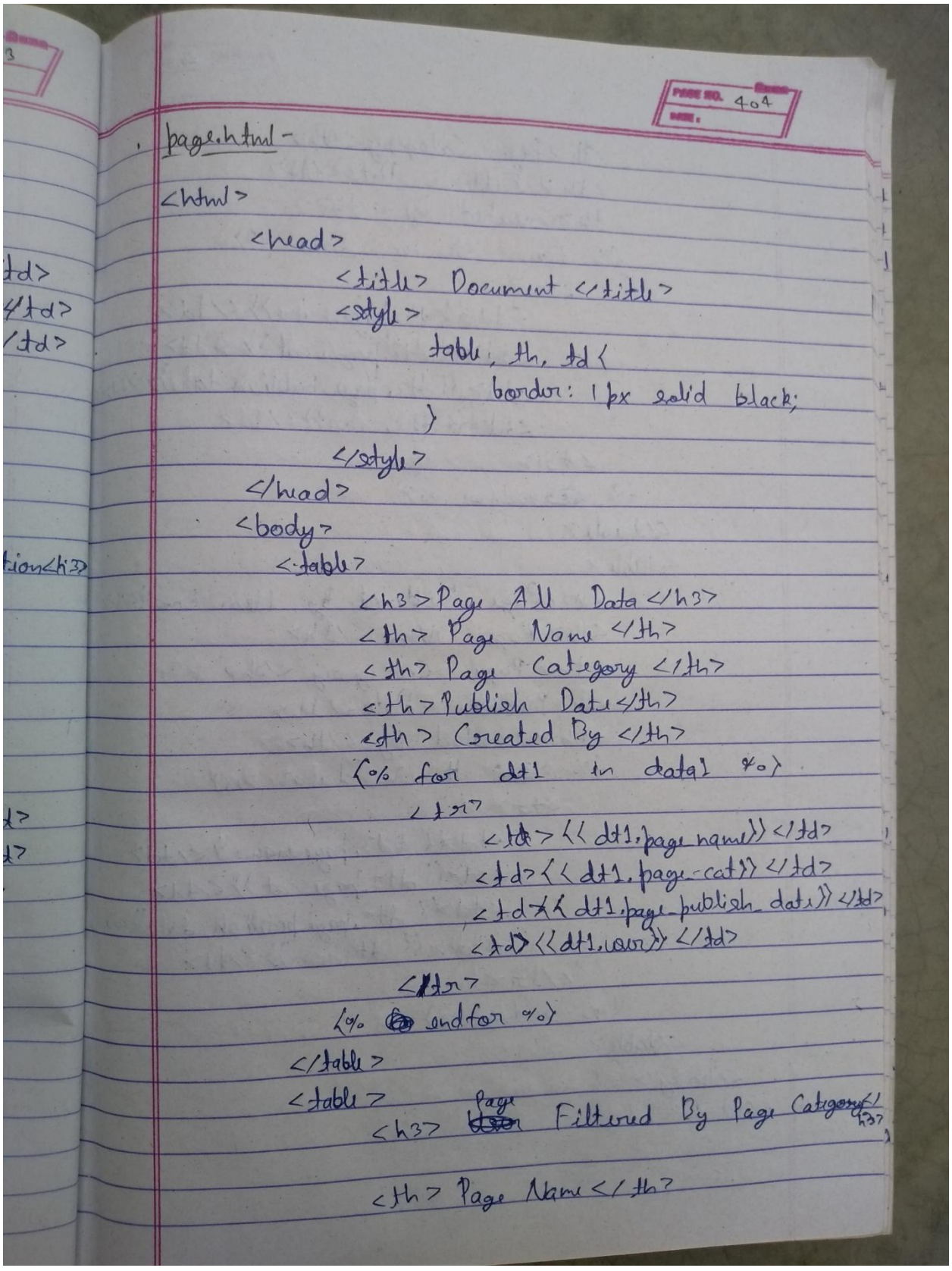
```
</th> Last Name </th>
<th> Email </th>
{% for dt1 in data1 %}
    <tr>
        <td>{{ dt1.username }} </td>
        <td>{{ dt1.first_name }} </td>
        <td>{{ dt1.last_name }}</td>
        <td>{{ dt1.email }} </td>
    </tr>
{% endfor %}
</table>
<table>
    <h3> User Filtered By Email </h3>
    <th> Username</th>
    <th> First Name </th>
    <th> Last Name</th>
    <th> Email </th>
    {% for dt2 in data2 %}
        <tr>
            <td>{{ dt2.username }} </td>
            <td>{{ dt2.first_name }} </td>
            <td>{{ dt2.last_name }} </td>
            <td>{{ dt2.email }} </td>
        </tr>
    {% endfor %}
</table>
<table>
    <h3> User Filtered By Page Category</h3>
    <th>Username </th>
    <th> First Name </th>
    <th> Last Name </th>
```



```
<th> Email </th>
{% for dt3 in data3 %}
  <tr>
    <td>{{ dt3.username }} </td>
    <td>{{ dt3.first_name }} </td>
    <td>{{ dt3.last_name }} </td>
    <td>{{ dt3.email }} </td>
  </tr>
{% endfor %}
</table>
<table>
  <h3> User Filtered By Post Publish Date </h3>
  <th> User Name </th>
  <th> First Name </th>
  <th> Last Name </th>
  <th> Email </th>
  {% for dt3 in data3 %}
    <tr>
      <td>{{ dt3.username }} </td>
      <td>{{ dt3.first_name }} </td>
      <td>{{ dt3.last_name }} </td>
      <td>{{ dt3.email }} </td>
    </tr>
  {% endfor %}
</table>
<table>
  <h3>User Filtered By Post Publish Date</h3>
  <th> Username </th>
  <th>First Name </th>
```

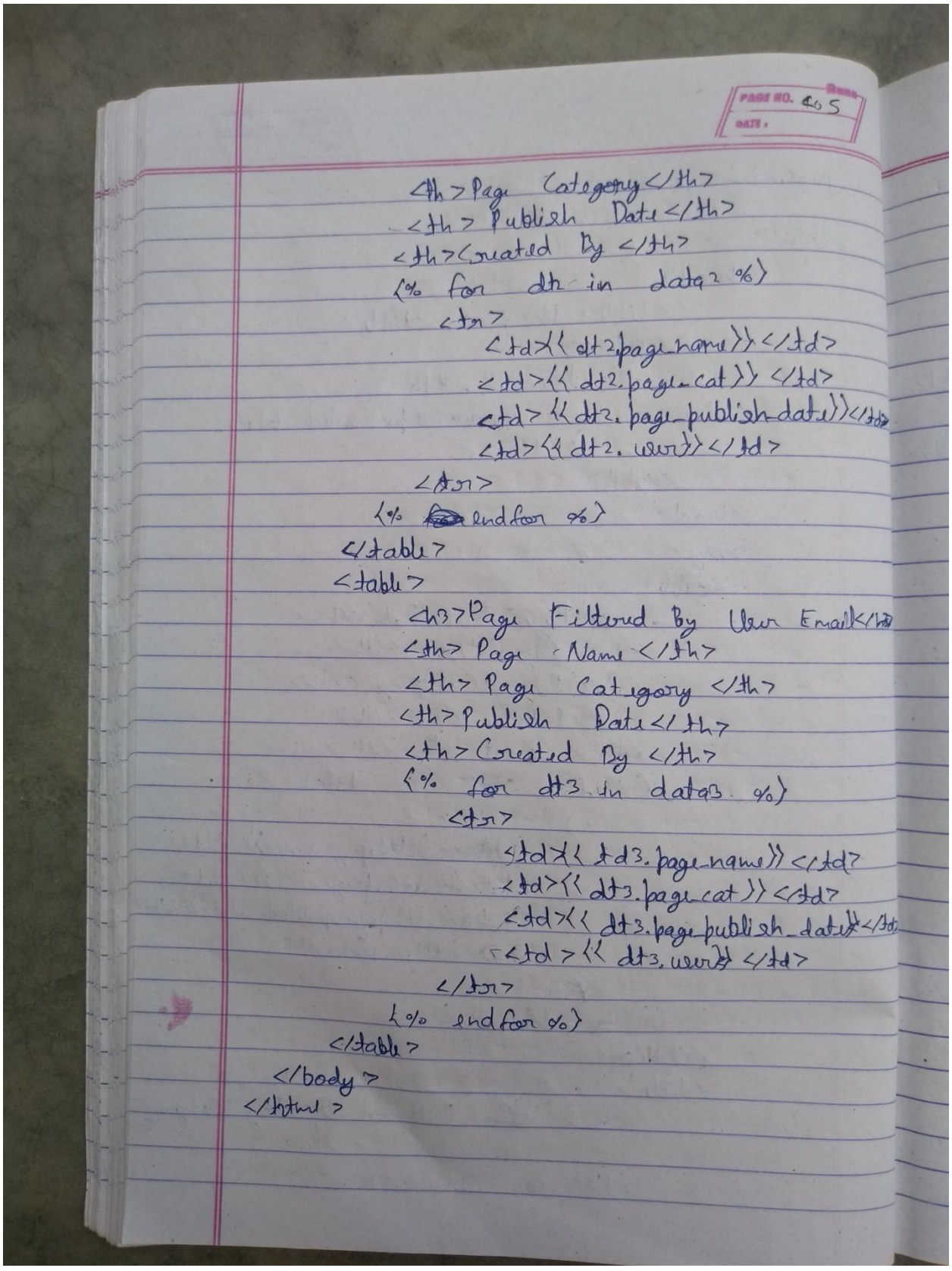


```
<th> Last Name </th>
<th> Email </th>
{% for dt4 in data4 %}
    <tr>
        <td>{{ td4.username }} </td>
        <td>{{ td4.first_name }}</td>
        <td>{{ td4.last_name }} </td>
        <td>{{ td4.email }} </td>
    </tr>
{% endfor %}
</table>
<table>
    <h3>User Filtered By Song Duration</h3>
    <th> Username</th>
    <th>First Name</th>
    <th> Last Name </th>
    <th> Email </th>
    {% for dt5 in data5 %}
      <tr>
        <td>{{ dt5.username }} </td>
        <td>{{ dt5.firstname }} </td>
        <td>{{ dt5.last_name }} </td>
        <td> {{ dt5.email }} </td>
      </tr>
    {% endfor %}
  </table>
</body>
</html>
```

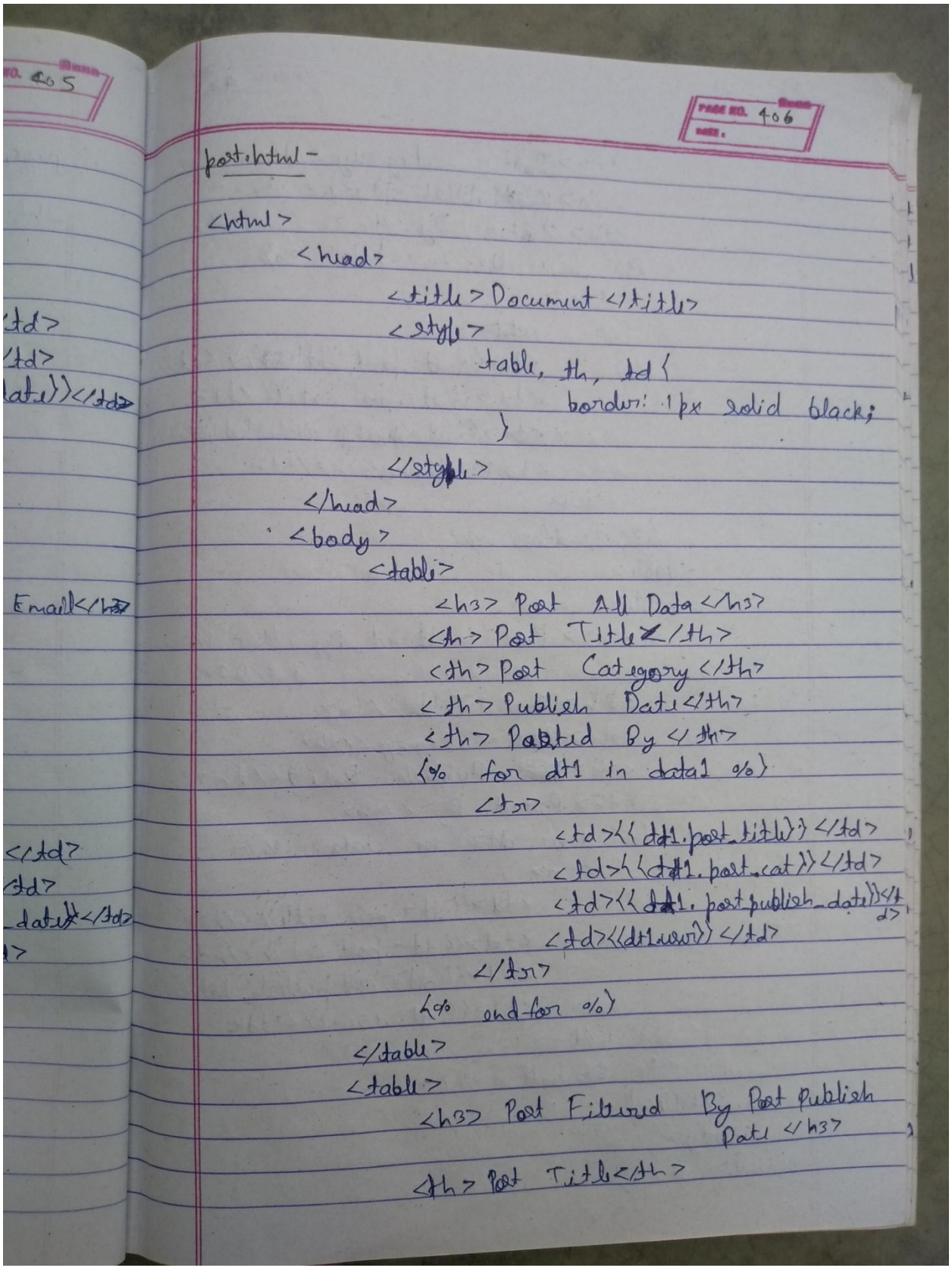


page.html -

```
<html>
    <head>
        <title> Document </title>
        <style>
            table, th, td {
                border: 1px solid black;
            }
        </style>
    </head>
    <body>
        <table>
            <h3> Page All Data </h3>
            <th> Page Name </th>
            <th> Page Category </th>
            <th> Publish Date </th>
            <th> Created By </th>
            {% for dt1 in data1 %}
            <tr>
                <td> {{ dt1.page_name }} </td>
                <td> {{ dt1.page_cat }} </td>
                <td> {{ dt1.page_publish_date }} </td>
                <td> {{ dt1.user }} </td>
            </tr>
            {% endfor %}
        </table>
        <table>
            <h3> Page Filtered By Page Category </h3>
            <th> Page Name </th>
```

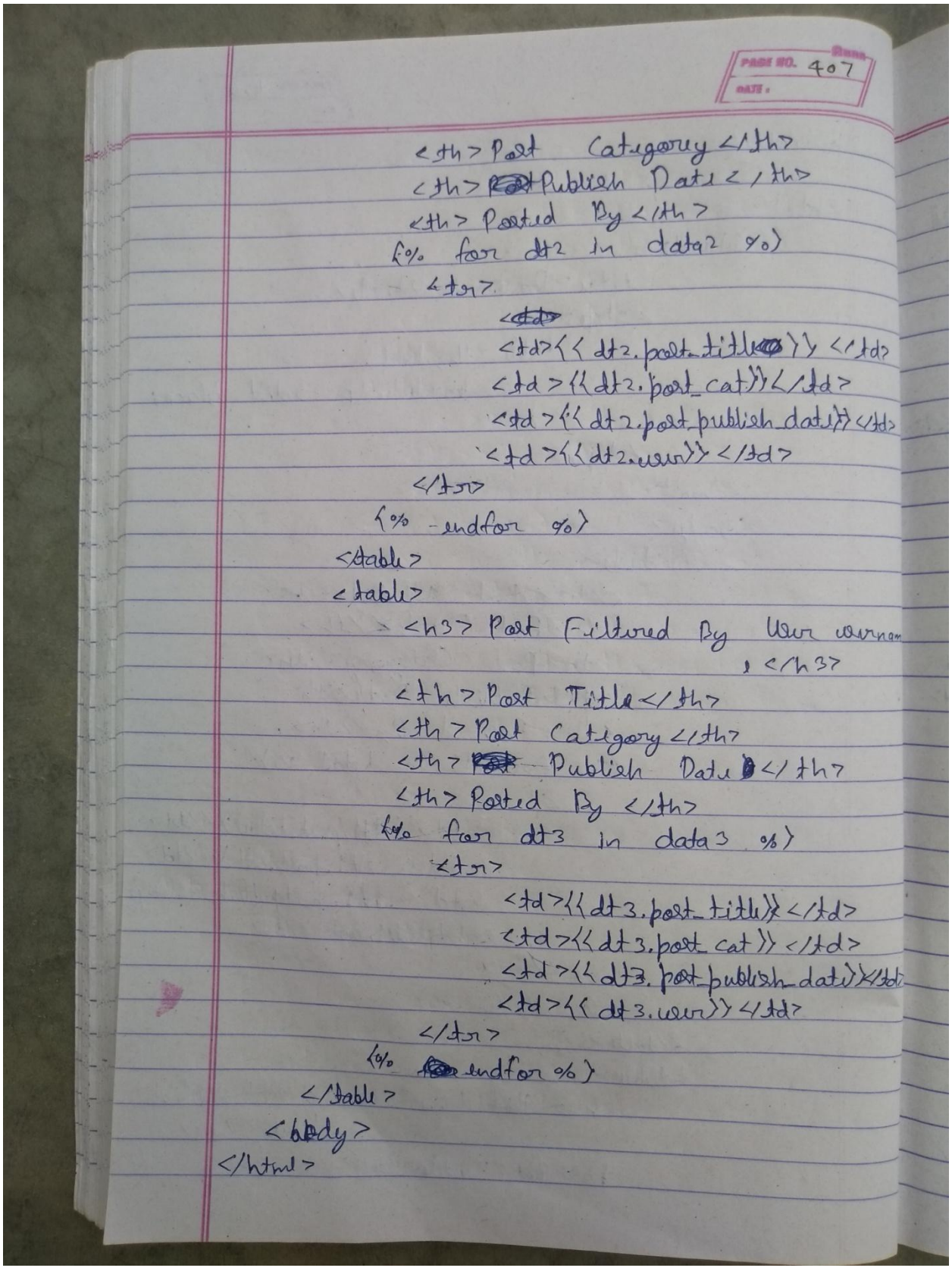


```
<th>Page Category</th>
<th>Publish Date</th>
<th>Created By</th>
{% for dt2 in data2 %}
  <tr>
    <td>{{ dt2.page_name }}</td>
    <td>{{ dt2.page_cat }}</td>
    <td>{{ dt2.page_publish_date }}</td>
    <td>{{ dt2.user }}</td>
  </tr>
{% endfor %}
</table>
<table>
  <th>Page Filtered By User Email</th>
  <th>Page Name</th>
  <th>Page Category</th>
  <th>Publish Date</th>
  <th>Created By</th>
  {% for dt3 in data3 %}
    <tr>
      <td>{{ dt3.page_name }}</td>
      <td>{{ dt3.page_cat }}</td>
      <td>{{ dt3.page_publish_date }}</td>
      <td>{{ dt3.user }}</td>
    </tr>
  {% endfor %}
</table>
</body>
</html>
```

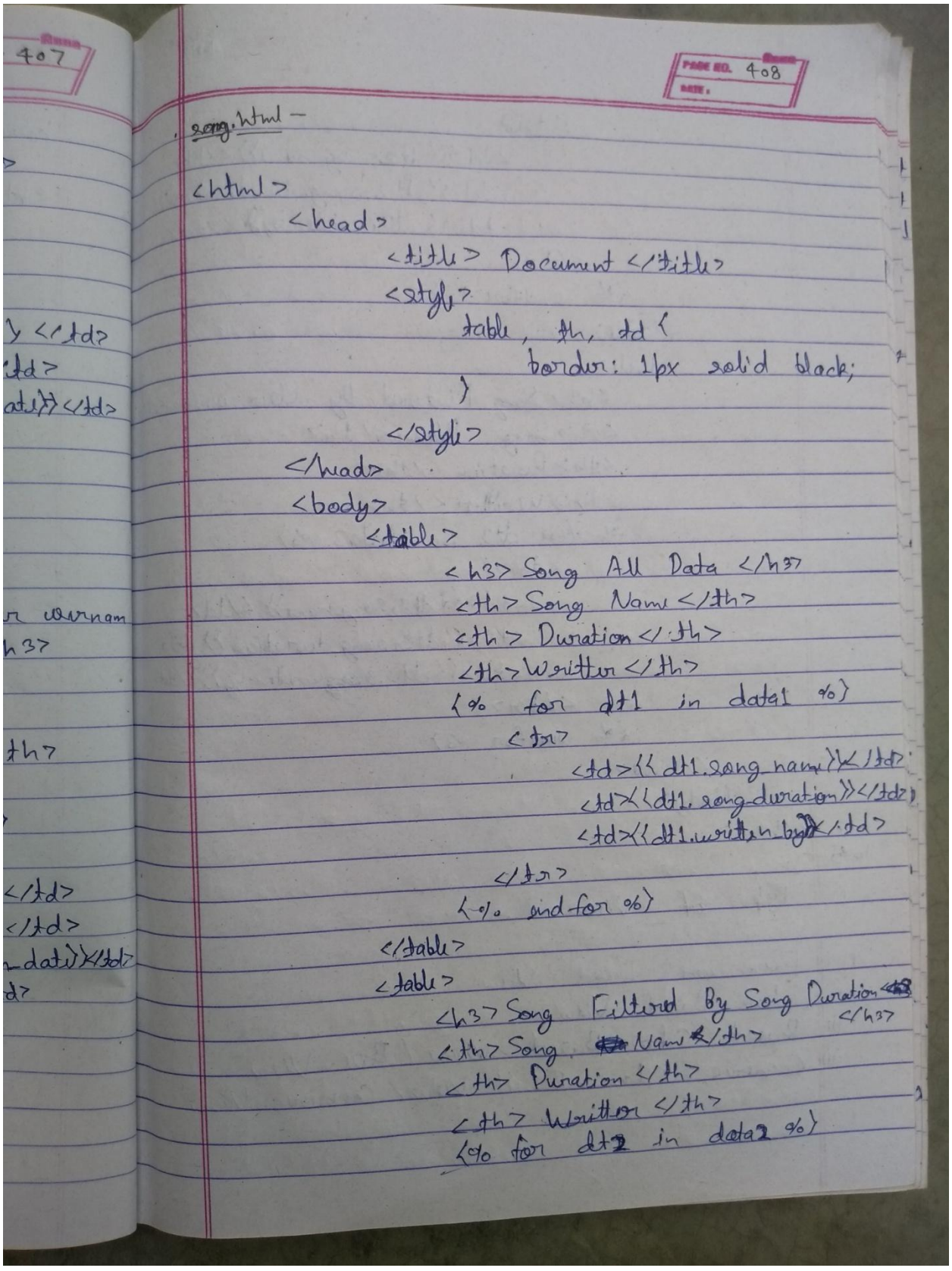


post.html -

```
<html>
    <head>
        <title> Document </title>
        <style>
            table, th, td {
                border: 1px solid black;
            }
        </style>
    </head>
    <body>
        <table>
            <h3> Post All Data </h3>
            <th> Post Title </th>
            <th> Post Category </th>
            <th> Publish Date </th>
            <th> Posted By </th>
            {% for dt1 in data1 %}
            <tr>
                <td>{{ dt1.post_title }} </td>
                <td>{{ dt1.post_cat }} </td>
                <td>{{ dt1.post_publish_date }} </td>
                <td>{{ dt1.user }} </td>
            </tr>
            {% endfor %}
        </table>
        <table>
            <h3> Post Filtered By Post Publish Date </h3>
            <th> Post Title </th>
```

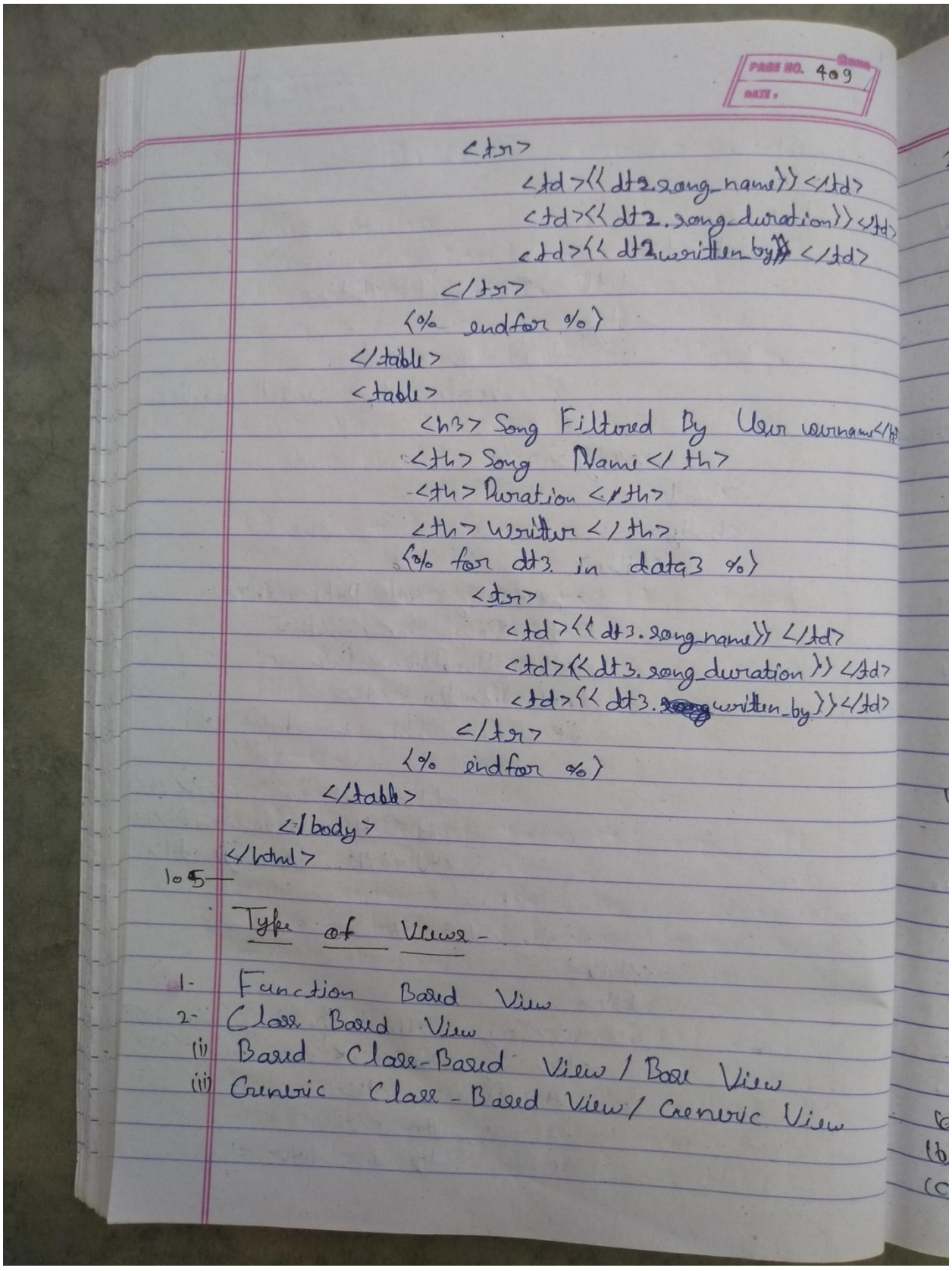


```
<th> Post Category </th>
<th> Publish Date </th>
<th> Posted By </th>
{% for dt2 in data2 %}
  <tr>
    <td>{{ dt2.post_title }} </td>
    <td>{{ dt2.post_cat }}</td>
    <td>{{ dt2.post_publish_date }}</td>
    <td>{{ dt2.user }} </td>
  </tr>
{% endfor %}
</table>
<table>
  <h3> Post Filtered By User username </h3>
  <th> Post Title </th>
  <th> Post Category </th>
  <th> Publish Date </th>
  <th> Posted By </th>
  {% for dt3 in data3 %}
    <tr>
      <td>{{ dt3.post_title }} </td>
      <td>{{ dt3.post_cat }} </td>
      <td>{{ dt3.post_publish_date }}</td>
      <td>{{ dt3.user }} </td>
    </tr>
  {% endfor %}
</table>
</body>
</html>
```



song.html –

```
<html>
    <head>
        <title> Document </title>
        <style>
            table, th, td {
                border: 1px solid black;
            }
        </style>
    </head>
    <body>
        <table>
            <h3> Song All Data </h3>
            <th> Song Name </th>
            <th> Duration </th>
            <th> Written </th>
            {% for dt1 in data1 %}
            <tr>
                <td>{{ dt1.song_name }}</td>
                <td>{{ dt1.song_duration }}</td>
                <td>{{ dt1.written_by }}</td>
            </tr>
            {% endfor %}
        </table>
        <table>
            <h3> Song Filtered By Song Duration </h3>
            <th> Song Name </th>
            <th> Duration </th>
            <th> Written </th>
            {% for dt2 in data2 %}
```

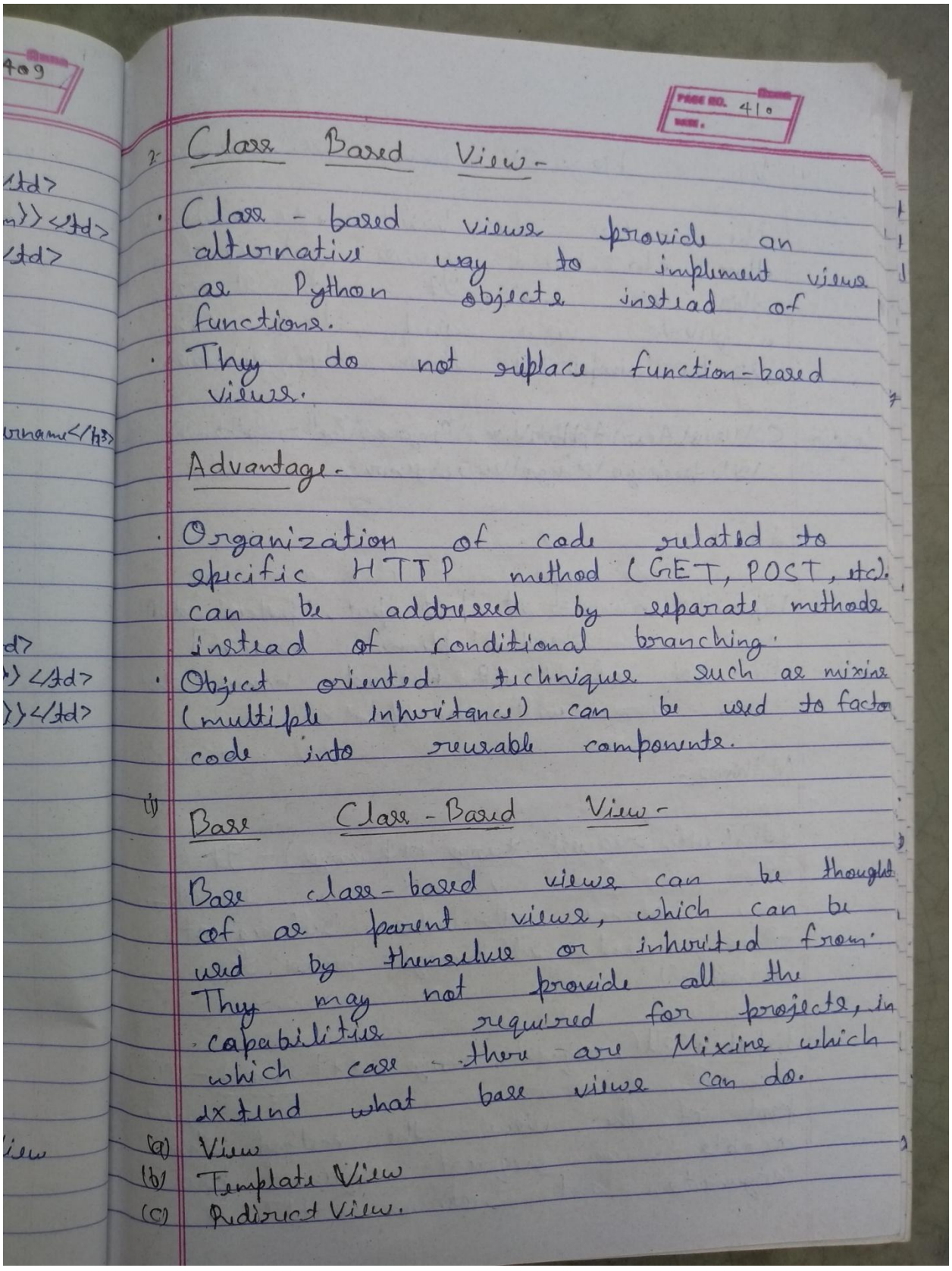


```
            <tr>
                <td>{{ dt2.song_name }}</td>
                <td>{{ dt2.song_duration }}</td>
                <td>{{ dt2.written_by }}</td>
            </tr>
        {% endfor %}
    </table>
    <table>
        <h3> Song Filtered By User username</h3>
        <th> Song Name </th>
        <th> Duration </th>
        <th> Writter </th>
        {% for dt3 in data3 %}
            <tr>
                <td>{{ dt3.song_name }} </td>
                <td>{{ dt3.song_duration }} </td>
                <td>{{ dt3.written_by }}</td>
            </tr>
        {% endfor %}
    </table>
</body>
</html>
```

105—

Type of Views -

1. Function Based View
2. Class Based View
   (i) Based Class-Based View / Base View
   (ii) Generic Class-Based View / Generic View



2. Class Based View -

- Class - based views provide an alternative way to implement views as Python objects instead of functions.
- They do not replace function-based views.

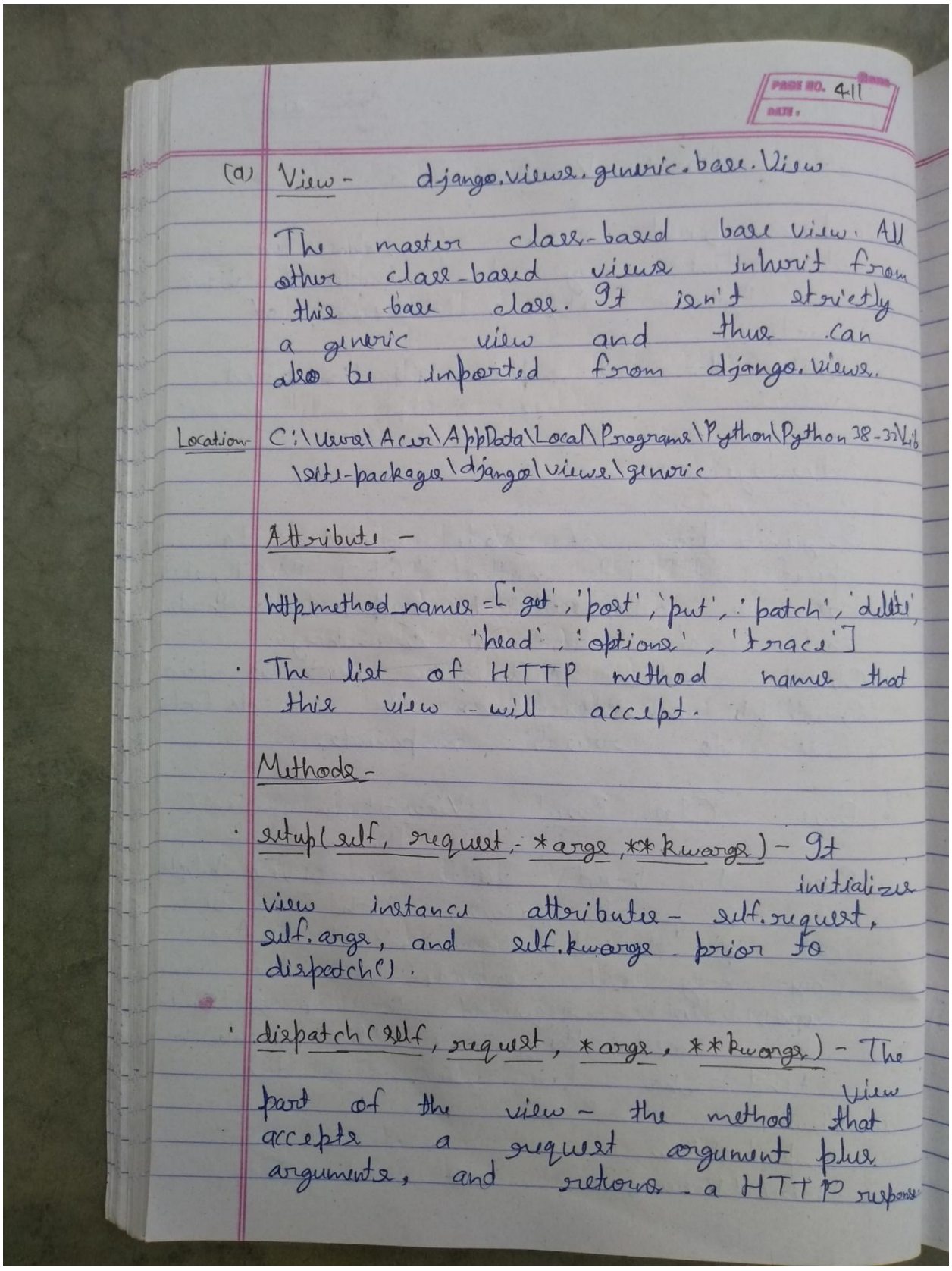
Advantage -

- Organization of code related to specific HTTP method (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

i) Base Class - Based View -

Base class-based views can be thought of as parent views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

(a) View
(b) Template View
(c) Redirect View.



PAGE NO. 411
DATE :

(a) View - django.views.generic.base.View

The master class-based base view. All other class-based views inherit from this base class. It isn't strictly a generic view and thus can also be imported from django.views.

Location - C:\Users\Acer\AppData\Local\Programs\Python\Python38-32\Lib\site-packages\django\views\generic

Attributes -

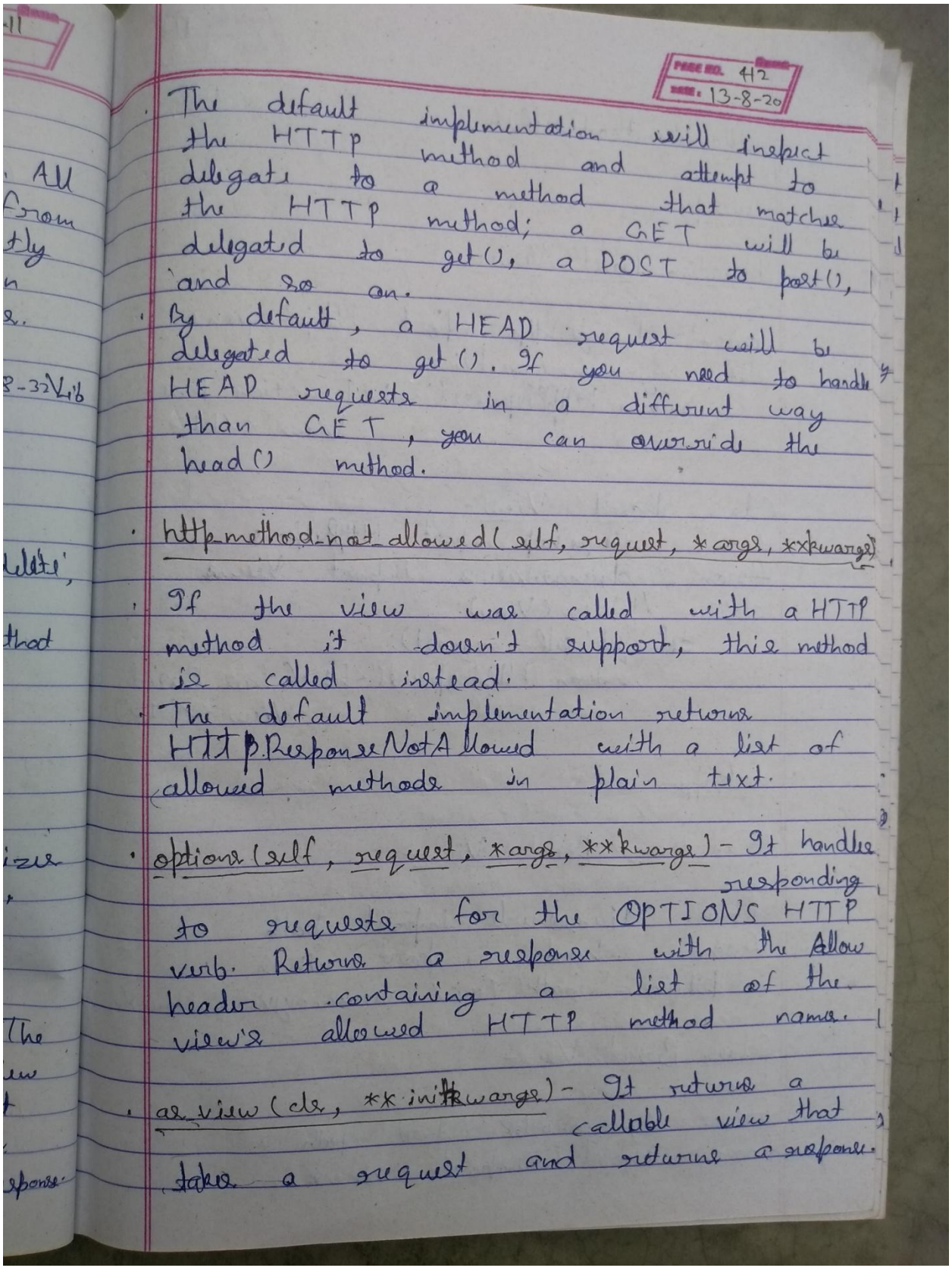
http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']

- The list of HTTP method names that this view will accept.

Methods -

- setup(self, request, *args, **kwargs) - It initializes view instance attributes - self.request, self.args, and self.kwargs prior to dispatch().

- dispatch(self, request, *args, **kwargs) - The view part of the view - the method that accepts a request argument plus arguments, and returns a HTTP response

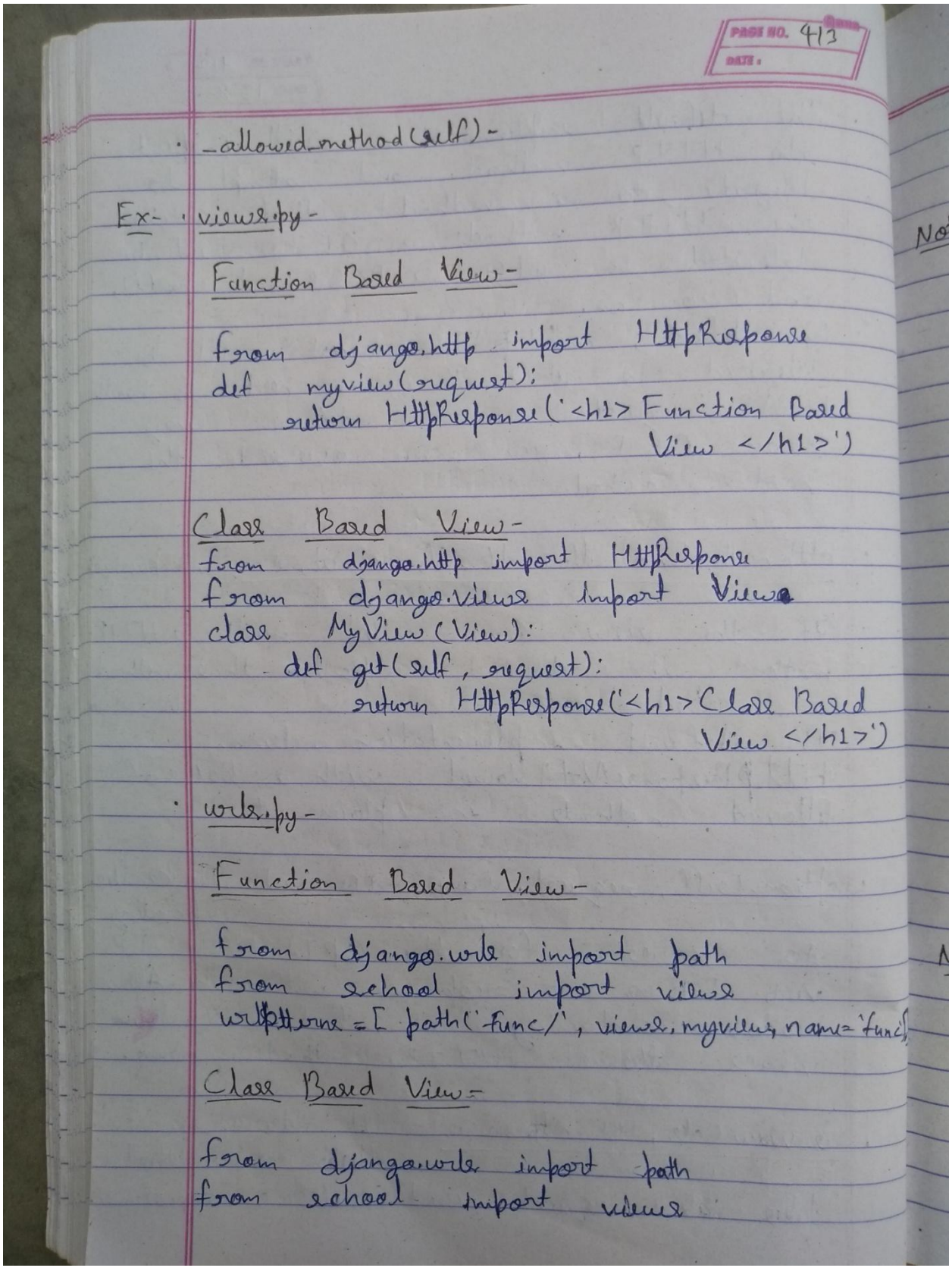


- The default implementation will inspect the HTTP method and attempt to delegate to a method that matches the HTTP method; a GET will be delegated to get(), a POST to post(), and so on.
- By default, a HEAD request will be delegated to get(). If you need to handle HEAD requests in a different way than GET, you can override the head() method.

- http_method_not_allowed(self, request, *args, **kwargs)
- If the view was called with a HTTP method it doesn't support, this method is called instead.
- The default implementation returns HttpResponseNotAllowed with a list of allowed methods in plain text.

- options(self, request, *args, **kwargs) - It handles responding to requests for the OPTIONS HTTP verb. Returns a response with the Allow header containing a list of the view's allowed HTTP method names.

- as_view(cls, **initkwargs) - It returns a callable view that takes a request and returns a response.



PAGE NO. 413
DATE :

- _allowed_method(self)-

Ex- · views.py-

Function Based View-

```
from django.http import HttpResponse
def myview(request):
    return HttpResponse('<h1> Function Based
                         View </h1>')
```

Class Based View-

```
from django.http import HttpResponse
from django.views import View
class MyView(View):
    def get(self, request):
        return HttpResponse('<h1> Class Based
                             View </h1>')
```

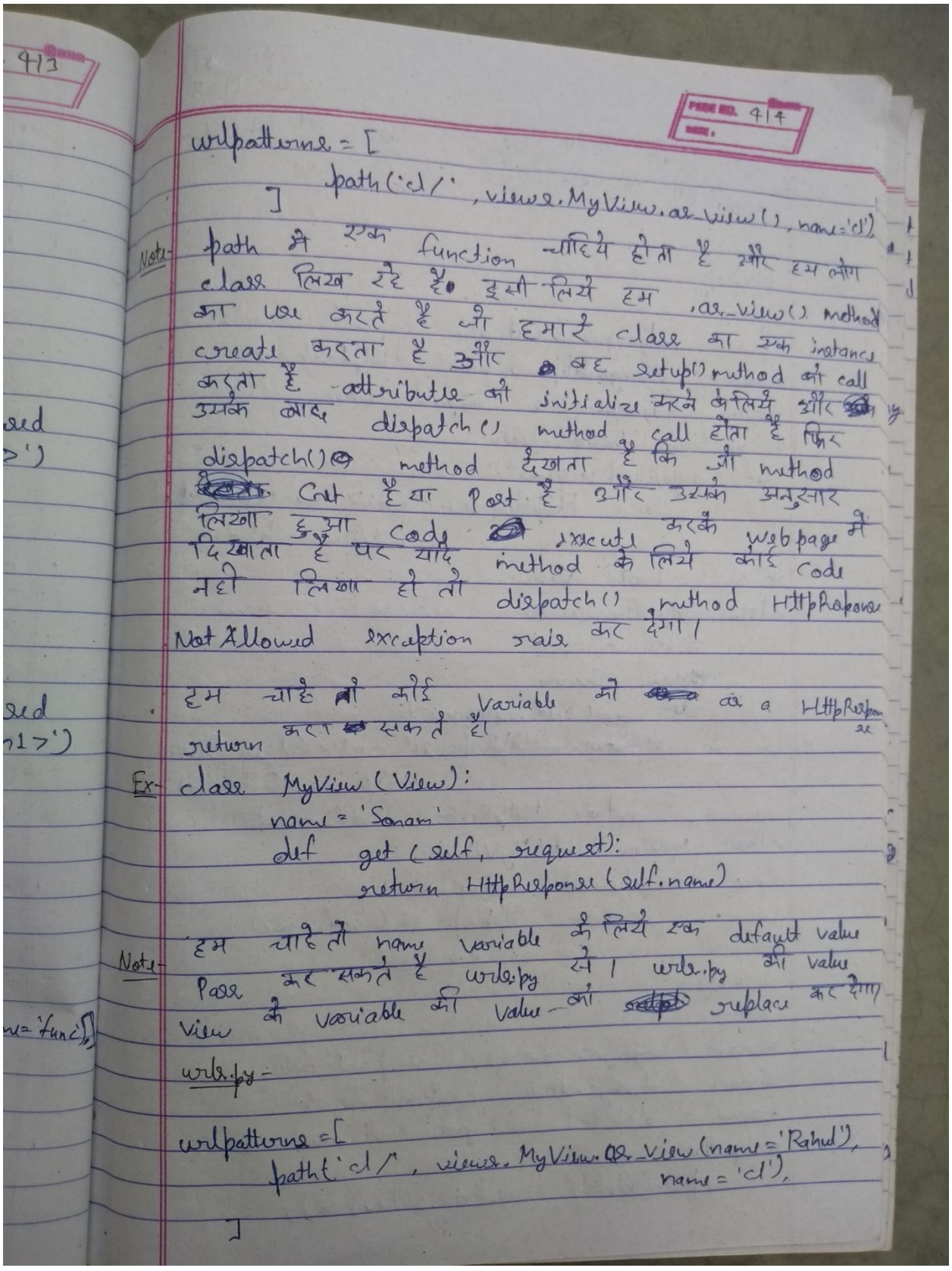
- urls.py-

Function Based View-

```
from django.urls import path
from school import views
urlpatterns = [ path('func/', views.myview, name='func'),
```

Class Based View-

```
from django.urls import path
from school import views
```



urlpatterns = [
    path('cl/', views.MyView.as_view(), name='cl')
]

Note- path में एक function चाहिये होता है और हम लोग class लिख रहे हैं। इसी लिये हम .as_view() method का use करते हैं जो हमारे class का एक instance create करता है और वह setup() method को call करता है attribute को initialize करने के लिये और उसके बाद dispatch() method call होता है फिर dispatch() method देखता है कि जो method है Get है या Post है और उसके अनुसार लिखा हुआ code execute करके webpage में दिखाता है पर यदि method के लिये कोई code नहीं लिखा है तो dispatch() method HttpResponse Not Allowed exception raise कर देगा।

हम चाहे तो कोई Variable को as a HttpResponse return कर सकते हैं

Ex- class MyView(View):
    name = 'Sonam'
    def get(self, request):
        return HttpResponse(self.name)

Note- हम चाहे तो name variable के लिये एक default value Pass कर सकते हैं urls.py से। urls.py की value View के variable की value को replace कर देगा।

urls.py -

urlpatterns = [
    path('cl/', views.MyView.as_view(name='Rahul'),
                name='cl'),
]

PAGE NO. 415
DATE : 14-8-20

Note- हम अपने किसी View class को inherit करके views भी कर सकते है।

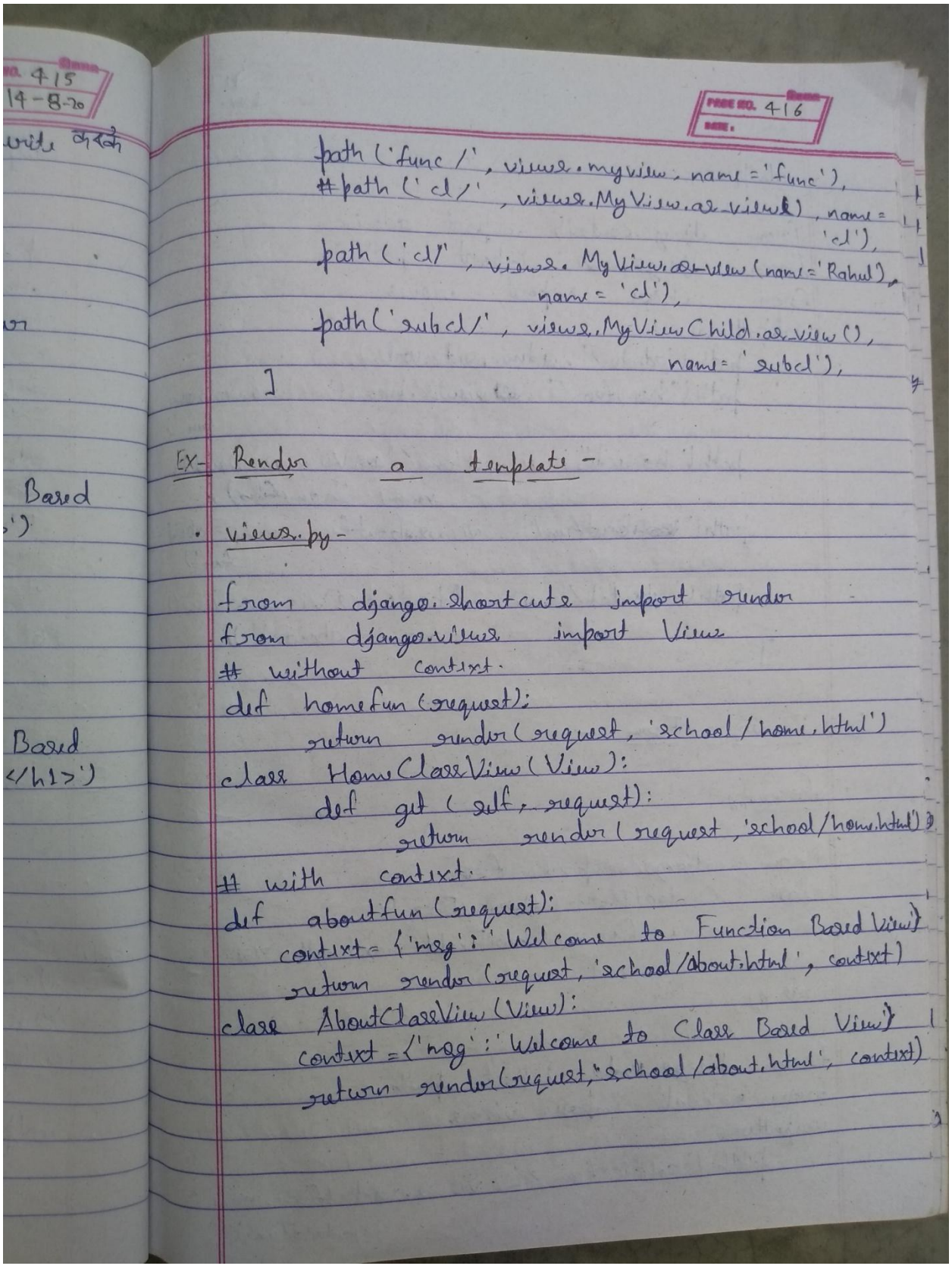
Ex- views.py -

```
from django.shortcuts import render
from django.http import HttpResponse
from django.views import View
#Function Based View
def myview(request):
    return HttpResponse('<h1> Function Based
                         View </h1>')

# class Based View
class MyView(View):
    # name = ''
    name = 'Sonam'
    def get(self, request):
        #return HttpResponse('<h1> Class Based
                             View - GET </h1>')
        return HttpResponse(self.name)
class MyViewChild(MyView):
    def get(self, request):
        return HttpResponse(self.name)
```

Ex-

urls.py -

```
from django.contrib import admin
from django.urls import path
from school import views
urlpatterns = [
    path('admin/', admin.site.urls),
```

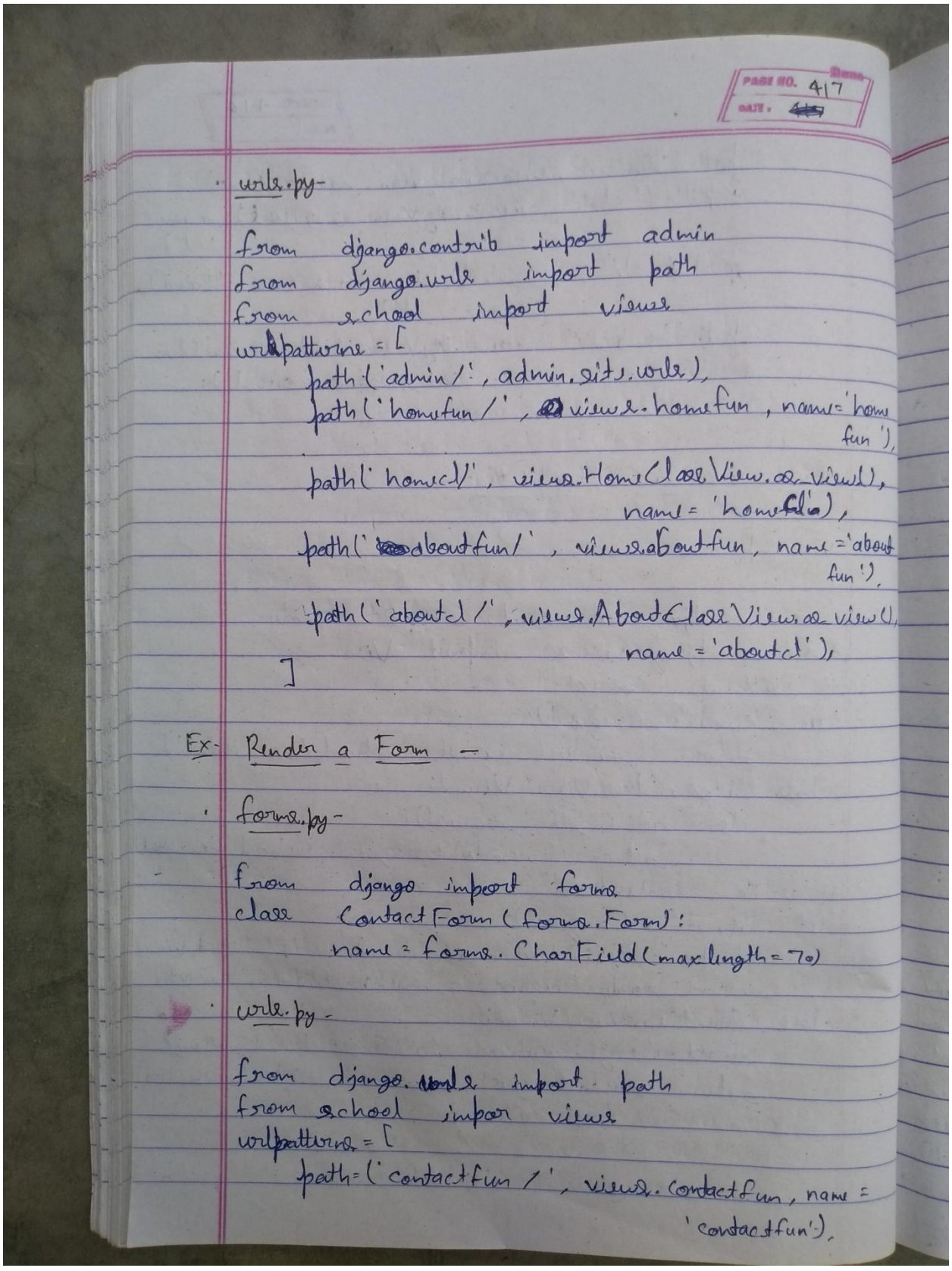


```
    path('func/', views.myview, name='func'),
    #path('cl/', views.MyView.as_view(), name=
                                        'cl'),
    path('cl/', views.MyView.as_view(name='Rahul'),
                                name='cl'),
    path('subcl/', views.MyViewChild.as_view(),
                                name='subcl'),
]
```

Ex- Render a template -

- views.py -

```
from django.shortcuts import render
from django.views import View
# without context.
def homefun(request):
    return render(request, 'school/home.html')
class HomeClassView(View):
    def get(self, request):
        return render(request, 'school/home.html')
# with context.
def aboutfun(request):
    context = {'msg': 'Welcome to Function Based View'}
    return render(request, 'school/about.html', context)
class AboutClassView(View):
    context = {'msg': 'Welcome to Class Based View'}
    return render(request, 'school/about.html', context)
```



PAGE NO. 417
DATE :

· urls.py -

```
from django.contrib import admin
from django.urls import path
from school import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('homefun/', views.homefun, name='homefun'),
    path('homecl/', views.HomeClassView.as_view(), name='homecl'),
    path('aboutfun/', views.aboutfun, name='aboutfun'),
    path('aboutcl/', views.AboutClassView.as_view(), name='aboutcl'),
]
```

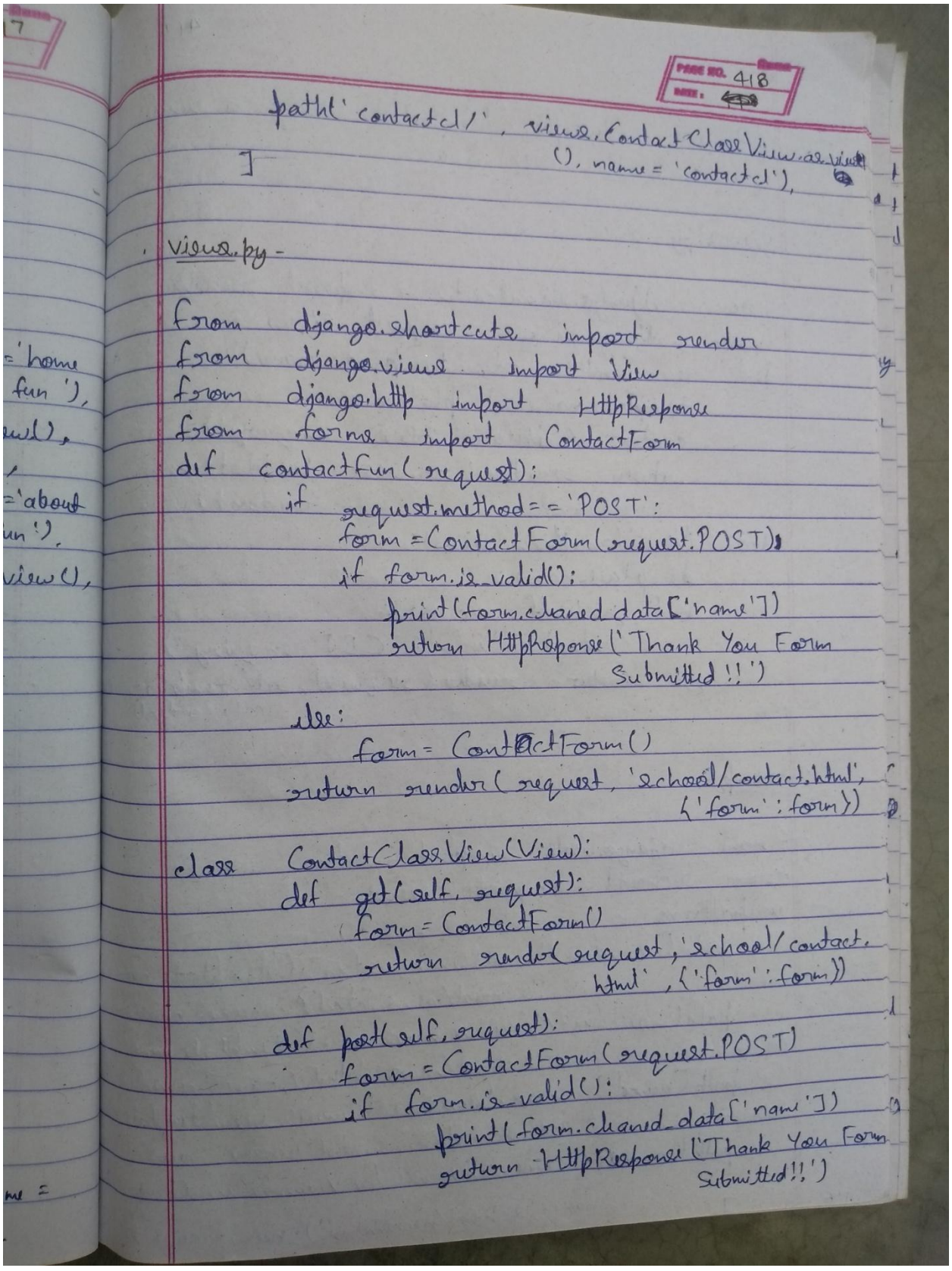
Ex- Render a Form -

· forms.py -

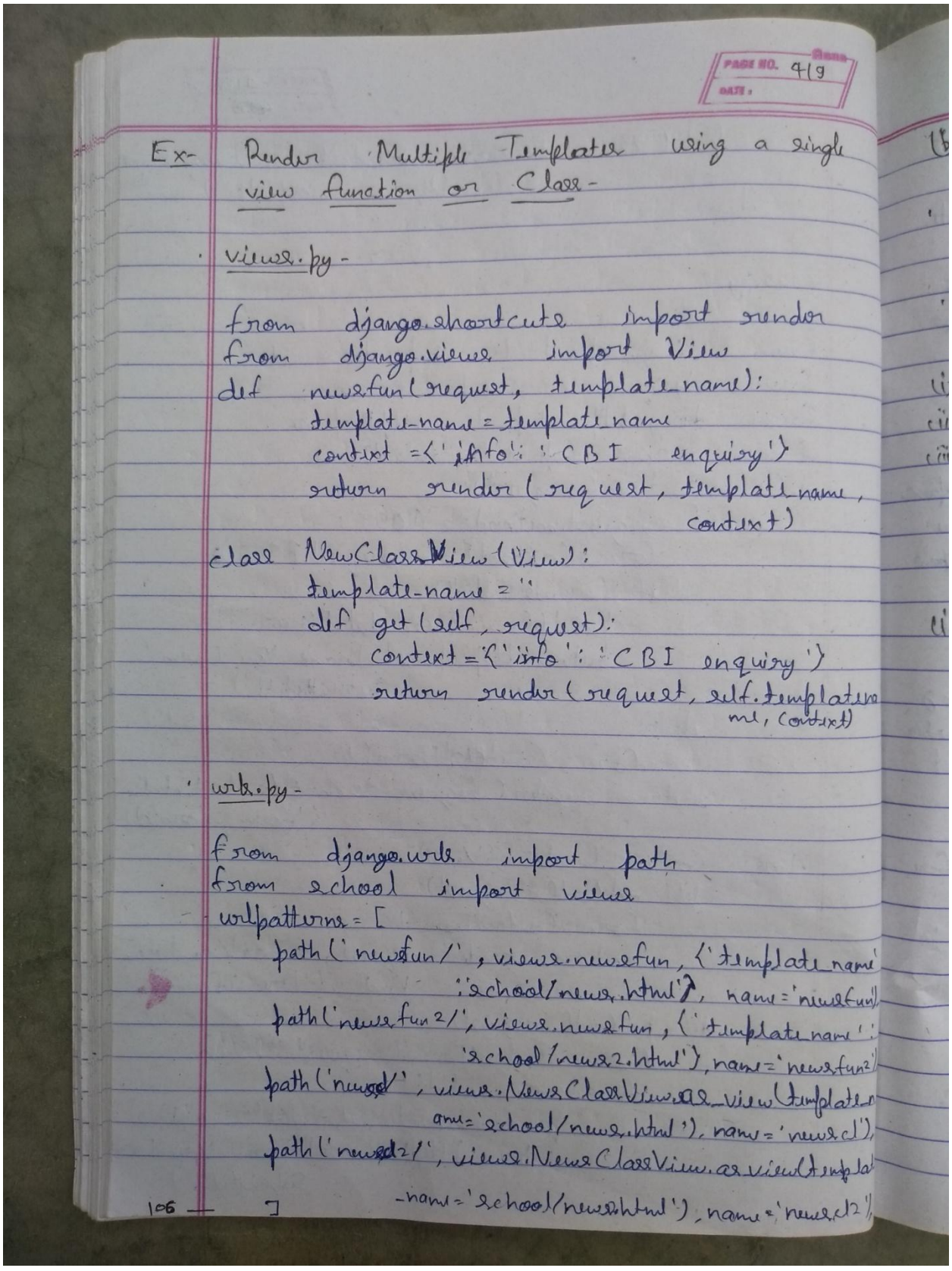
```
from django import forms
class ContactForm(forms.Form):
    name = forms.CharField(max_length=70)
```

· urls.py -

```
from django.urls import path
from school import views
urlpatterns = [
    path('contactfun/', views.contactfun, name='contactfun'),
```



```
    path('contactcl/', views.ContactClassView.as_view
                    (), name = 'contactcl'),
]

views.py -

from django.shortcuts import render
from django.views import View
from django.http import HttpResponse
from forms import ContactForm
def contactfun(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            print(form.cleaned_data['name'])
            return HttpResponse('Thank You Form
                                Submitted!!')
    else:
        form = ContactForm()
    return render(request, 'school/contact.html',
                                {'form': form})

class ContactClassView(View):
    def get(self, request):
        form = ContactForm()
        return render(request, 'school/contact.
                        html', {'form': form})

    def post(self, request):
        form = ContactForm(request.POST)
        if form.is_valid():
            print(form.cleaned_data['name'])
            return HttpResponse('Thank You Form
                                Submitted!!')
```



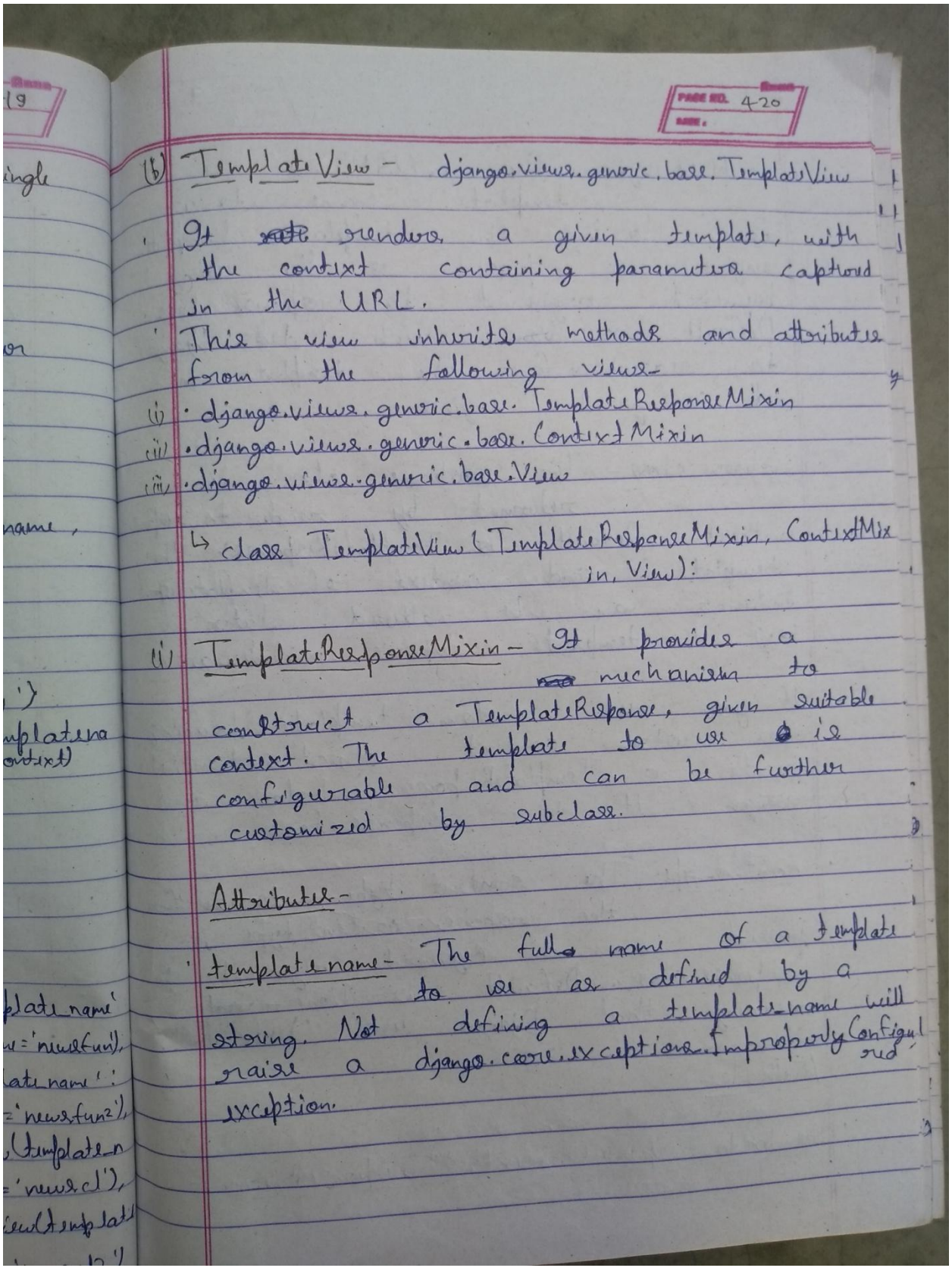
Ex- Render Multiple Templates using a single view function or Class-

- views.py -

```
from django.shortcuts import render
from django.views import View
def newsfun(request, template_name):
    template_name = template_name
    context = {'info': 'CBI enquiry'}
    return render(request, template_name,
                  context)
class NewsClassView(View):
    template_name = ''
    def get(self, request):
        context = {'info': 'CBI enquiry'}
        return render(request, self.template_name,
                      context)
```

- urls.py -

```
from django.urls import path
from school import views
urlpatterns = [
    path('newsfun/', views.newsfun, {'template_name':
        'school/news.html'}, name='newsfun'),
    path('newsfun2/', views.newsfun, {'template_name':
        'school/news2.html'}, name='newsfun2'),
    path('newscl/', views.NewsClassView.as_view(template_
        name='school/news.html'), name='newscl'),
    path('newscl2/', views.NewsClassView.as_view(templa
        -name='school/news2.html'), name='newscl2'),
]
```



(b) TemplateView - django.views.generic.base.TemplateView

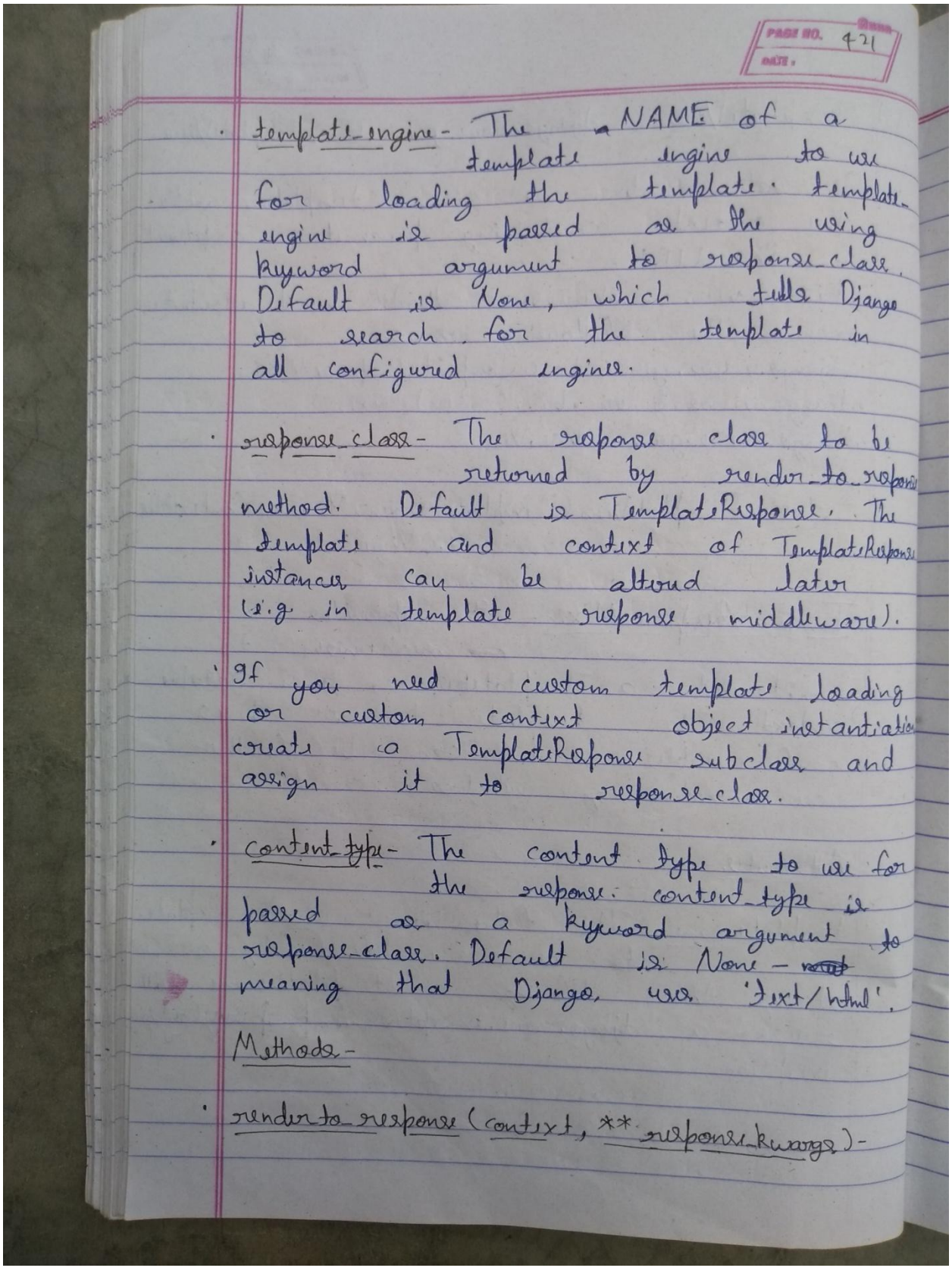
- It renders a given template, with the context containing parameters captured in the URL.
- This view inherits methods and attributes from the following views-
  (i) · django.views.generic.base.TemplateResponseMixin
  (ii) · django.views.generic.base.ContextMixin
  (iii) · django.views.generic.base.View

↳ class TemplateView(TemplateResponseMixin, ContextMixin, View):

(i) TemplateResponseMixin - It provides a mechanism to construct a TemplateResponse, given suitable context. The template to use is configurable and can be further customized by subclass.

Attributes-

- template_name - The full name of a template to use as defined by a string. Not defining a template_name will raise a django.core.exceptions.ImproperlyConfigured exception.



PAGE NO. 421
DATE :

- template_engine - The NAME of a template engine to use for loading the template. template_engine is passed as the using keyword argument to response_class. Default is None, which tells Django to search for the template in all configured engines.

- response_class - The response class to be returned by render_to_response method. Default is TemplateResponse. The template and context of TemplateResponse instances can be altered later (e.g. in template response middleware).

- If you need custom template loading or custom context object instantiation create a TemplateResponse subclass and assign it to response_class.

- content_type - The content type to use for the response. content_type is passed as a keyword argument to response_class. Default is None - meaning that Django uses 'text/html'.

Methods -

- render_to_response(context, **response_kwargs) -

PAGE NO. 422
DATE: 18-8-20

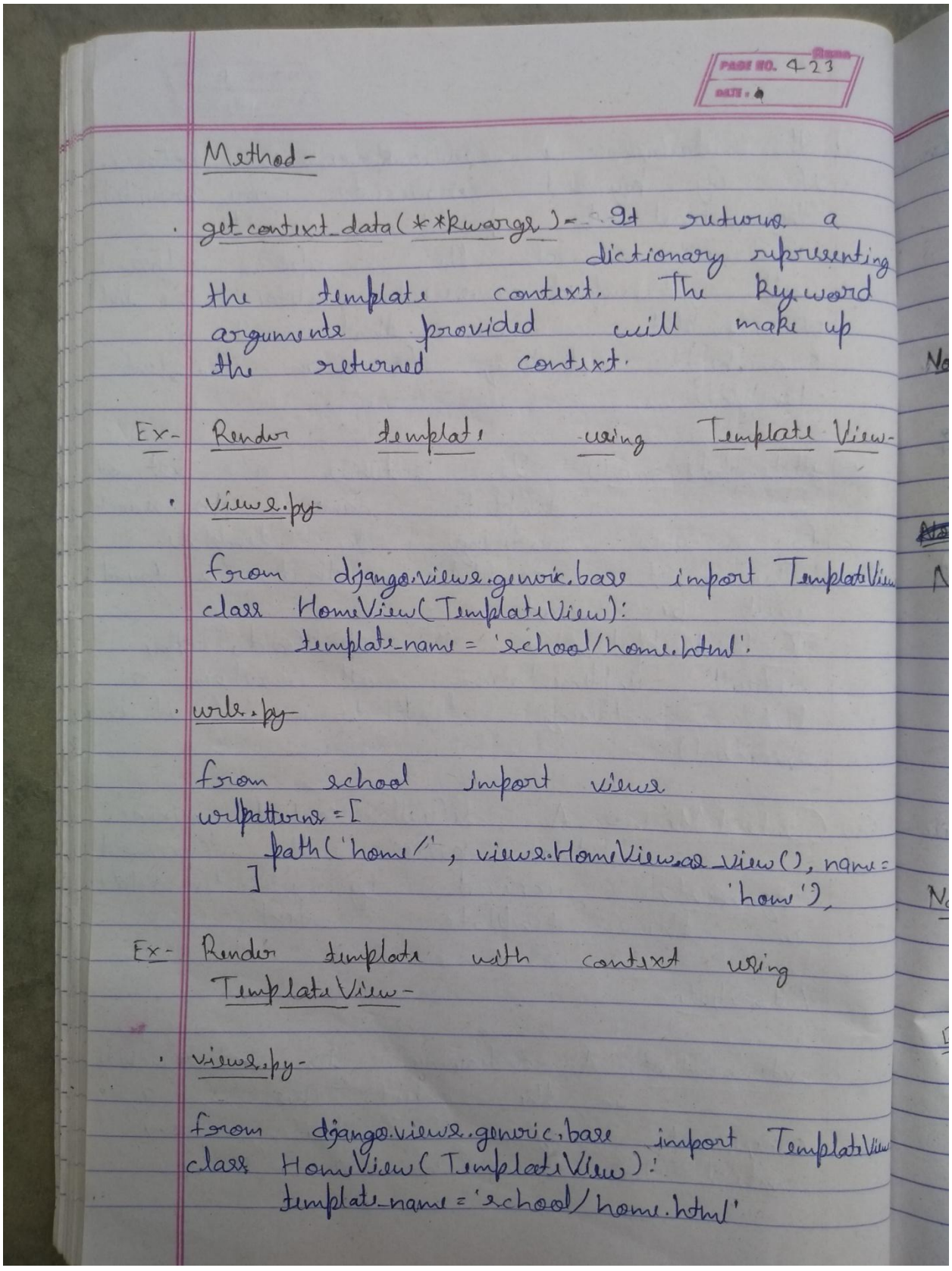
- It returns a `rdf.response_class` instance.
- If any keyword arguments are provided, they will be passed to the constructor of the response class.
- Calls `get_template_names()` to obtain the list of template names that will be searched looking for an existent template.

- `get_template_names()` - It returns a list of template names to search for when rendering the template. The first template that is found will be used.
- If `template_name` is specified, the default implementation will return a list containing `template_name` (if it is specified).

(ii) `ContextMixin` - A default context mixin that passes the keyword arguments received by `get_context_data()` as the template context.

Attributes -

- `extra_context` - A dictionary to ~~include~~ include in the context. This is a convenient way of specifing some context in `as_view()`.



PAGE NO. 423
DATE :

Method -

- get_context_data(**kwargs) - It returns a dictionary representing the template context. The keyword arguments provided will make up the returned context.

Ex- Render template using TemplateView -

- views.py

```
from django.views.generic.base import TemplateView
class HomeView(TemplateView):
    template_name = 'school/home.html'
```

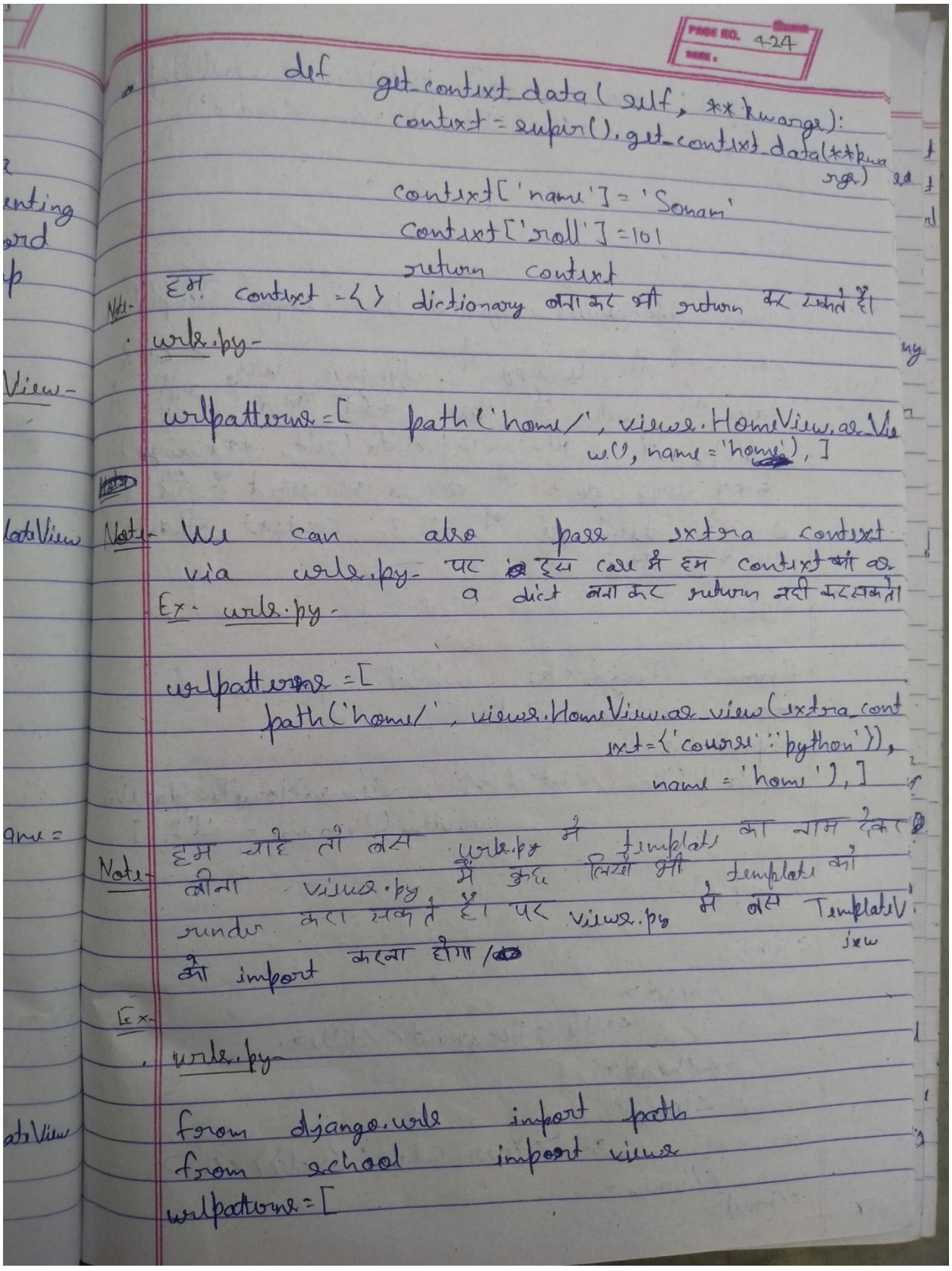
- urls.py

```
from school import views
urlpatterns = [
    path('home/', views.HomeView.as_view(), name='how'),
]
```

Ex- Render template with context using TemplateView -

- views.py -

```
from django.views.generic.base import TemplateView
class HomeView(TemplateView):
    template_name = 'school/home.html'
```



```
def get_context_data(self, **kwargs):
    context = super().get_context_data(**kwargs)
    context['name'] = 'Sonam'
    context['roll'] = 101
    return context
```

Note- इस context = {} dictionary बना कर भी return कर सकते है।

- urls.py-

```
urlpatterns = [ path('home/', views.HomeView.as_view(), name='home'), ]
```

Note- We can also pass extra context via urls.py- पर इस case में इस context को as a dict बना कर return नही कर सकते।

Ex- urls.py-

```
urlpatterns = [
    path('home/', views.HomeView.as_view(extra_context={'course':'python'}),
         name='home'), ]
```

Note- हम चाहे तो बस urls.py में template का नाम देकर बीना views.py में कुछ लिखे भी template को render करा सकते है। पर views.py में बस TemplateView को import करना होगा।

Ex-

- urls.py-

```
from django.urls import path
from school import views
urlpatterns = [
```

```
    path('index/', views.TemplateView.as_view(template_name
        ='school/home.html'), name='index'),
]
```

- views.py -

```
from django.views.generic.base import TemplateView
```

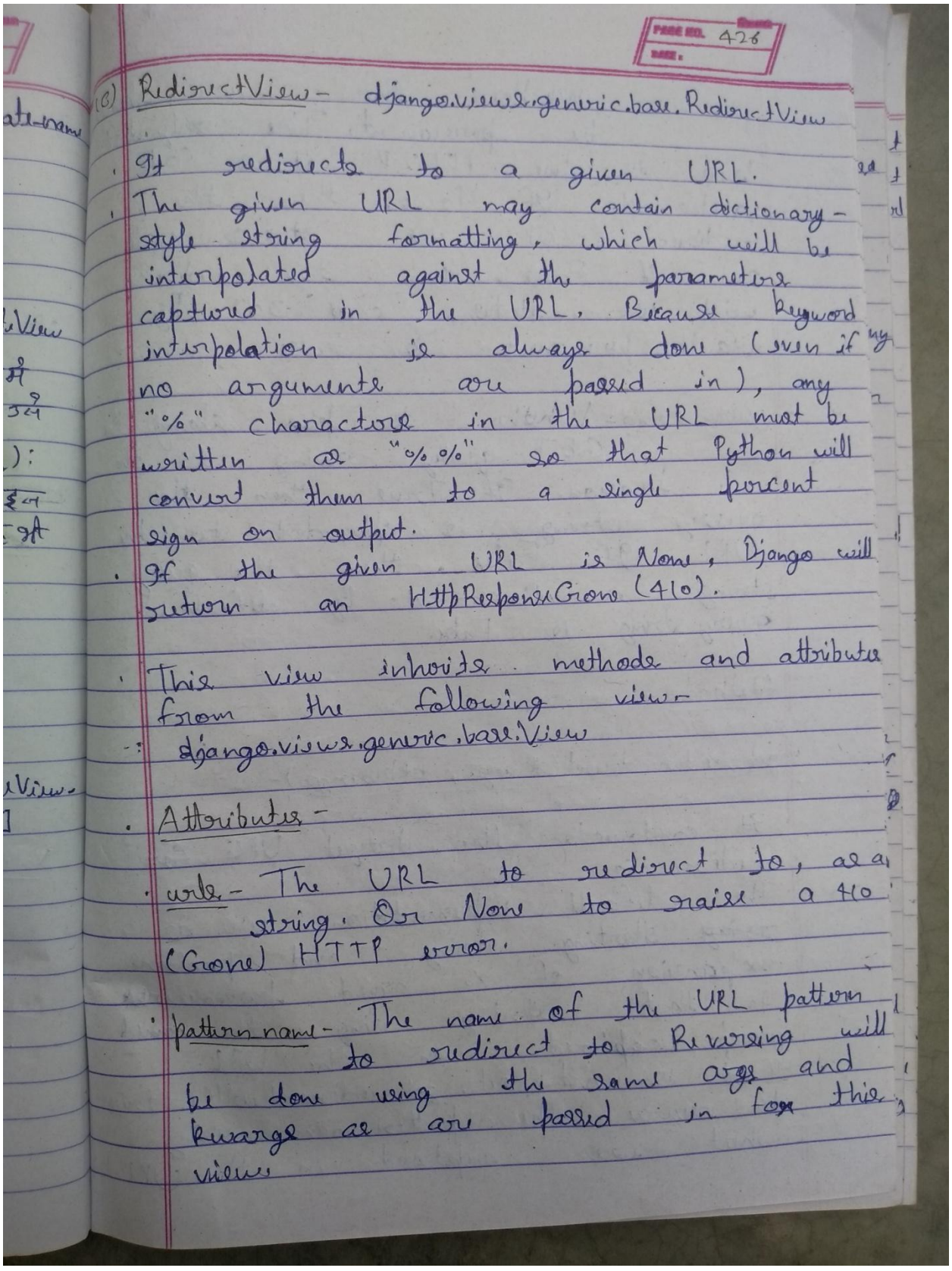
Note- हम चाहे तो urls.py में dynamic url के रूप में data को urls से pass कर सकते है और उसे views.py में get_context_data(self, **kwargs): के **kwargs dict से access कर सकते है और इस उसने template में as a context वापस भी use कर सकते है

Ex:- urls.py -

```
from django.urls import path
from school import views
urlpatterns = [
    path('home/<int:d>/', views.HomeTemplateView
                .as_view(), name='d'), ]
```

- home.html -

```
<html>
    <head>
        <title> Document </title>
    </head>
    <body>
        <h2> Your Class: {{d}} </h2>
    </body>
</html>
```



(c) RedirectView - django.views.generic.base.RedirectView

- It redirects to a given URL.
- The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is always done (even if no arguments are passed in), any "%" characters in the URL must be written as "%%" so that Python will convert them to a single percent sign on output.
- If the given URL is None, Django will return an HttpResponseGone (410).

- This view inherits methods and attributes from the following view -
  - django.views.generic.base.View

- Attributes -

- url - The URL to redirect to, as a string. Or None to raise a 410 (Gone) HTTP error.

- pattern_name - The name of the URL pattern to redirect to. Reversing will be done using the same args and kwargs as are passed in for this view.

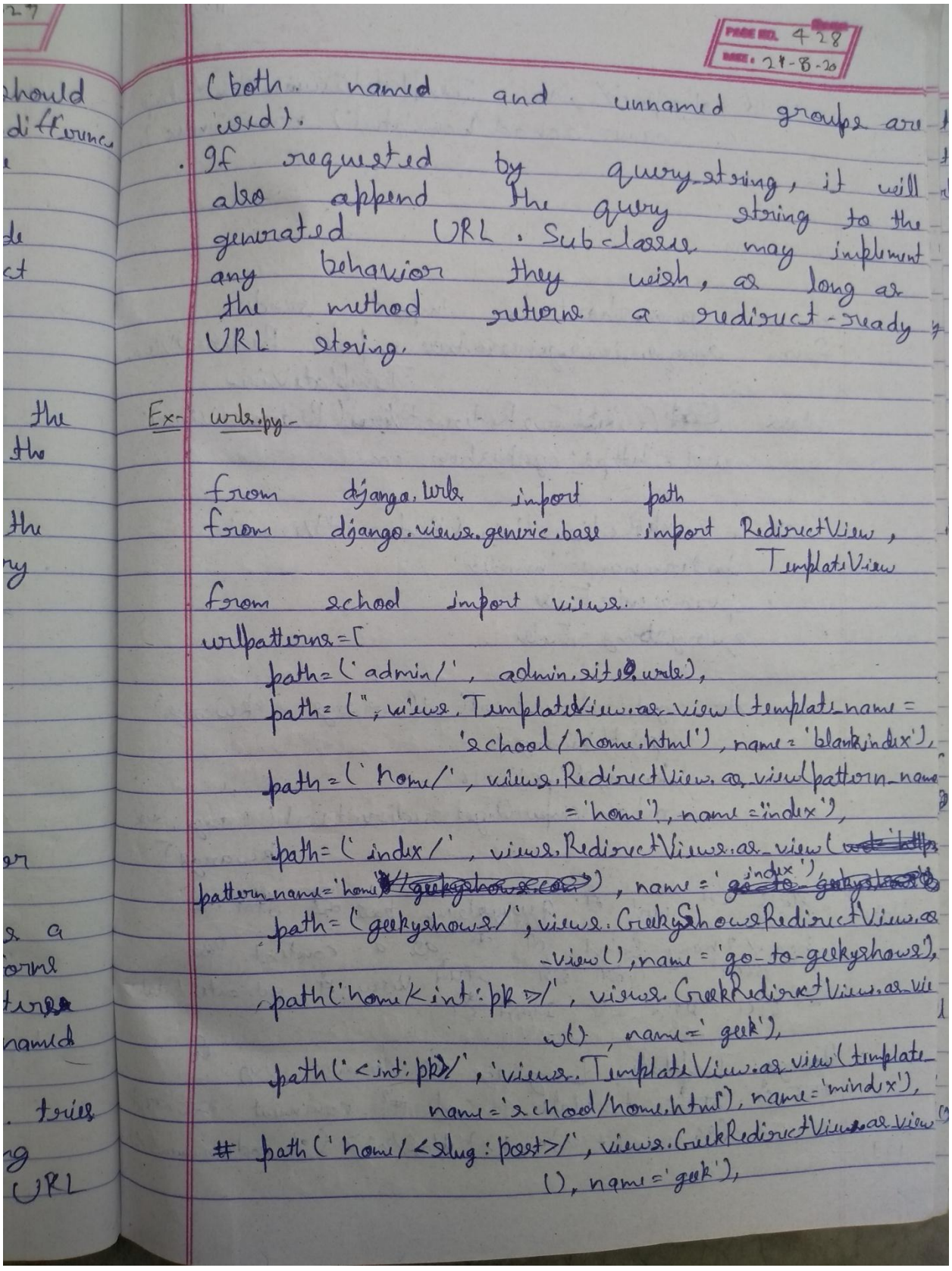
- permanent - Whether the redirect should be permanent. The only difference here is the HTTP status code returned. If True, then the redirect will use status code 301. If False, then the redirect will use status code 302. By default, permanent is False.

- query_string - Whether to pass along the GET query string to the new location. If True, then the query string is appended to the URL. If False, then the query string is discarded. By default, query_string is False.

Methods -

- get_redirect_url(*args, **kwargs) -

  - It constructs the target URL for redirection.
  - The default implementation uses as a ~~string~~ starting string and performs expansion of % named parameters in that string using the named groups captured in the URL.
  - If url is not set, get_redirect_url() tries to reverse the pattern_name using what was captured in the URL



(both named and unnamed groups are used).

- If requested by query_string, it will also append the query string to the generated URL. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

Ex- urls.py -

```
from django.urls import path
from django.views.generic.base import RedirectView, TemplateView
from school import views
urlpatterns=[
    path('admin/', admin.site.urls),
    path('', views.TemplateView.as_view(template_name='school/home.html'), name='blankindex'),
    path('home/', views.RedirectView.as_view(pattern_name='home'), name='index'),
    path('index/', views.RedirectView.as_view(pattern_name='home'), name='index'),
    path('geekyshows/', views.GeekyShowsRedirectView.as_view(), name='go-to-geekyshows'),
    path('home/<int:pk>/', views.GeekRedirectView.as_view(), name='geek'),
    path('<int:pk>/', views.TemplateView.as_view(template_name='school/home.html'), name='mindx'),
# path('home/<slug:post>/', views.GeekRedirectView.as_view(), name='geek'),
```

PAGE NO. 429
DATE :

```
#path('<slug:post>', views.TemplateView.view( template
    name='school/home.html'), name='index'),
]
```

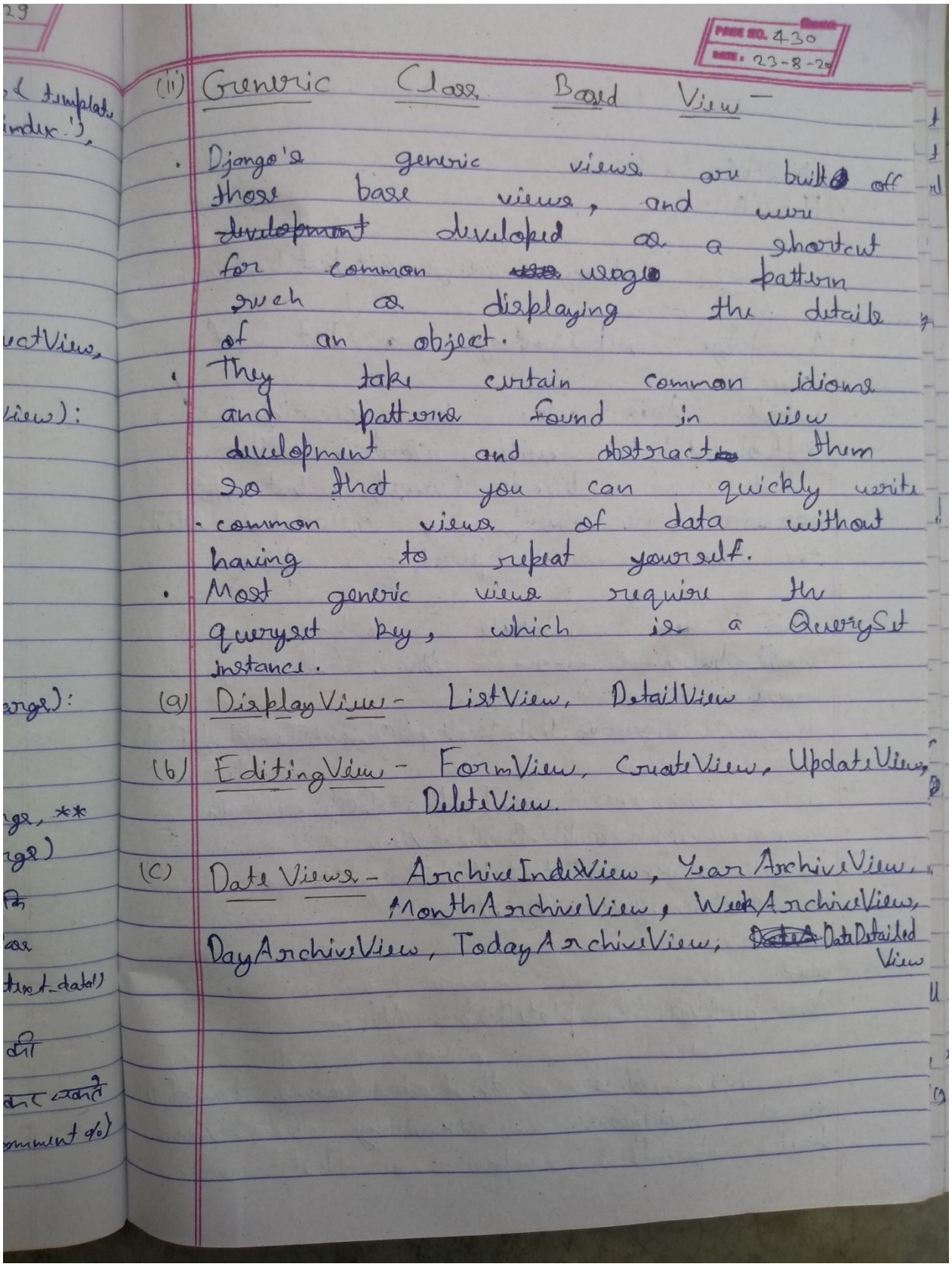
views.py-

```
from django.shortcuts import render
from django.views.generic.base import RedirectView,
                                      TemplateView
class GeekyshowsRedirectView(RedirectView):
    url = 'https://geekyshows.com'

class GeekRedirectView(RedirectView):
    pattern_name = 'mindex'
    permanent = False
    query_string = False

    def get_redirect_url(self, *args, **kwargs):
        print(kwargs)
        kwargs['pk'] = 16
        return super().get_redirect_url(*args, **
                                        kwargs)
```

Note- अगर हम url में query string pass करें जैसे कि home/?name=Rahul तो वह as a context pass हो जाता है जिसे हम अपने view में get_context_data() के द्वारा प्राप्त कर सकते हैं।

Django Template Language में comment tag की मदद से अपने template के code को comment कर सकते हैं। Ex- {% comment %}<h1>Hello</h1>{% endcomment %}

109 —



PAGE NO. 430
DATE: 23-8-2

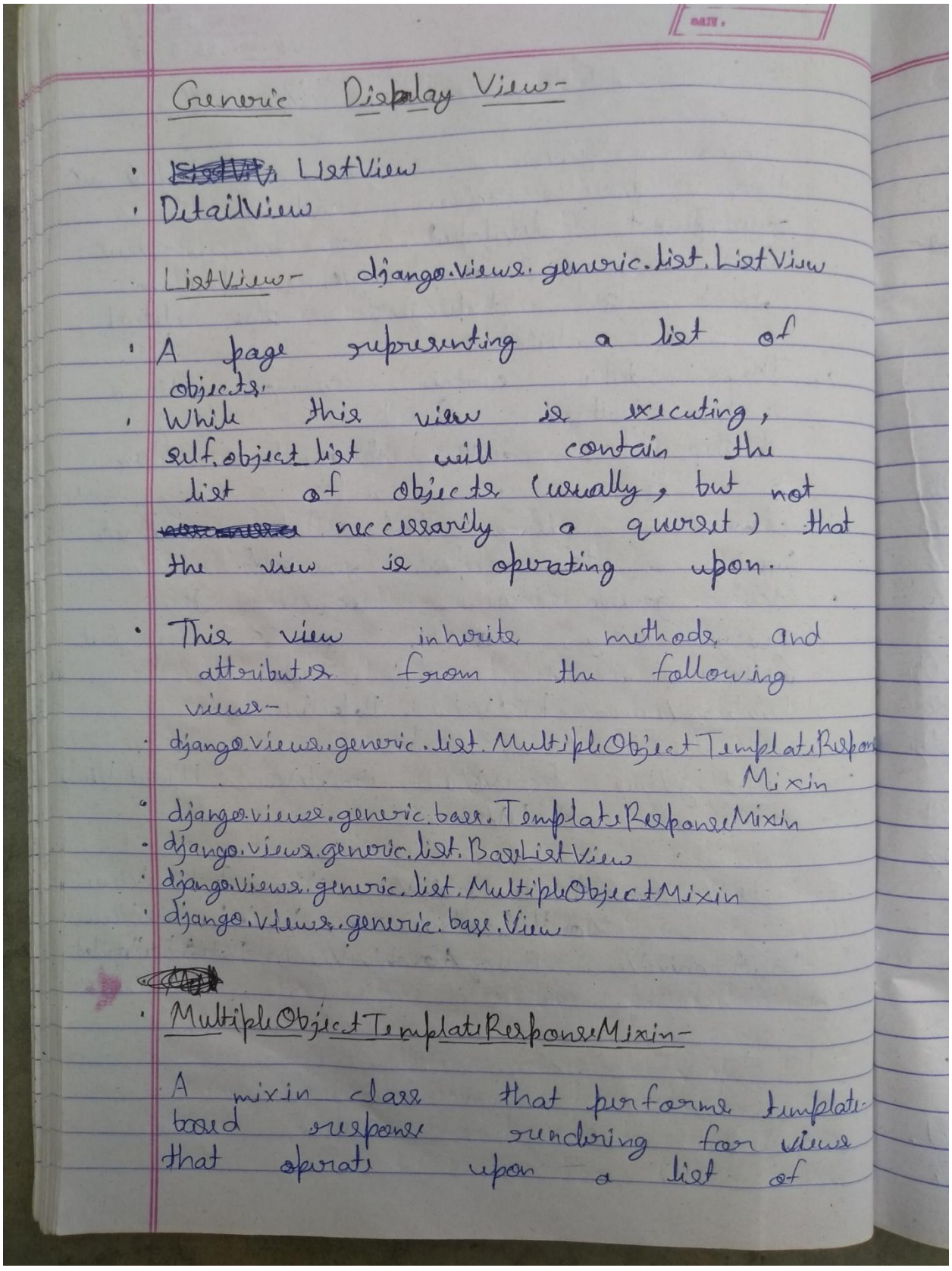
(ii) Generic Class Based View -

- Django's generic views are built off those base views, and were developed as a shortcut for common usage pattern such as displaying the details of an object.
- They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.
- Most generic views require the queryset key, which is a QuerySet instance.

(a) DisplayView - ListView, DetailView

(b) EditingView - FormView, CreateView, UpdateView, DeleteView.

(c) Date Views - ArchiveIndexView, YearArchiveView, MonthArchiveView, WeekArchiveView, DayArchiveView, TodayArchiveView, DateDetailView



Generic Display View-

- ~~ListView~~ ListView
- DetailView

ListView- django.views.generic.list.ListView

- A page representing a list of objects.
- While this view is executing, self.object_list will contain the list of objects (usually, but not ~~necessarily~~ necessarily a queryset) that the view is operating upon.
- This view inherits methods and attributes from the following views-
  - django.views.generic.list.MultipleObjectTemplateResponseMixin
  - django.views.generic.base.TemplateResponseMixin
  - django.views.generic.list.BaseListView
  - django.views.generic.list.MultipleObjectMixin
  - django.views.generic.base.View

- MultipleObjectTemplateResponseMixin-

A mixin class that performs template based response rendering for views that operate upon a list of

object instances. Requires that the view it is mixed with provides self.object_list, the list of object instances that the view is operating on. self.object_list may be but be, but is not required to be, a QuerySet.

- This inherits methods and attributes from the following views-
  django.views.generic.base.TemplateResponseMixin

Attribute-

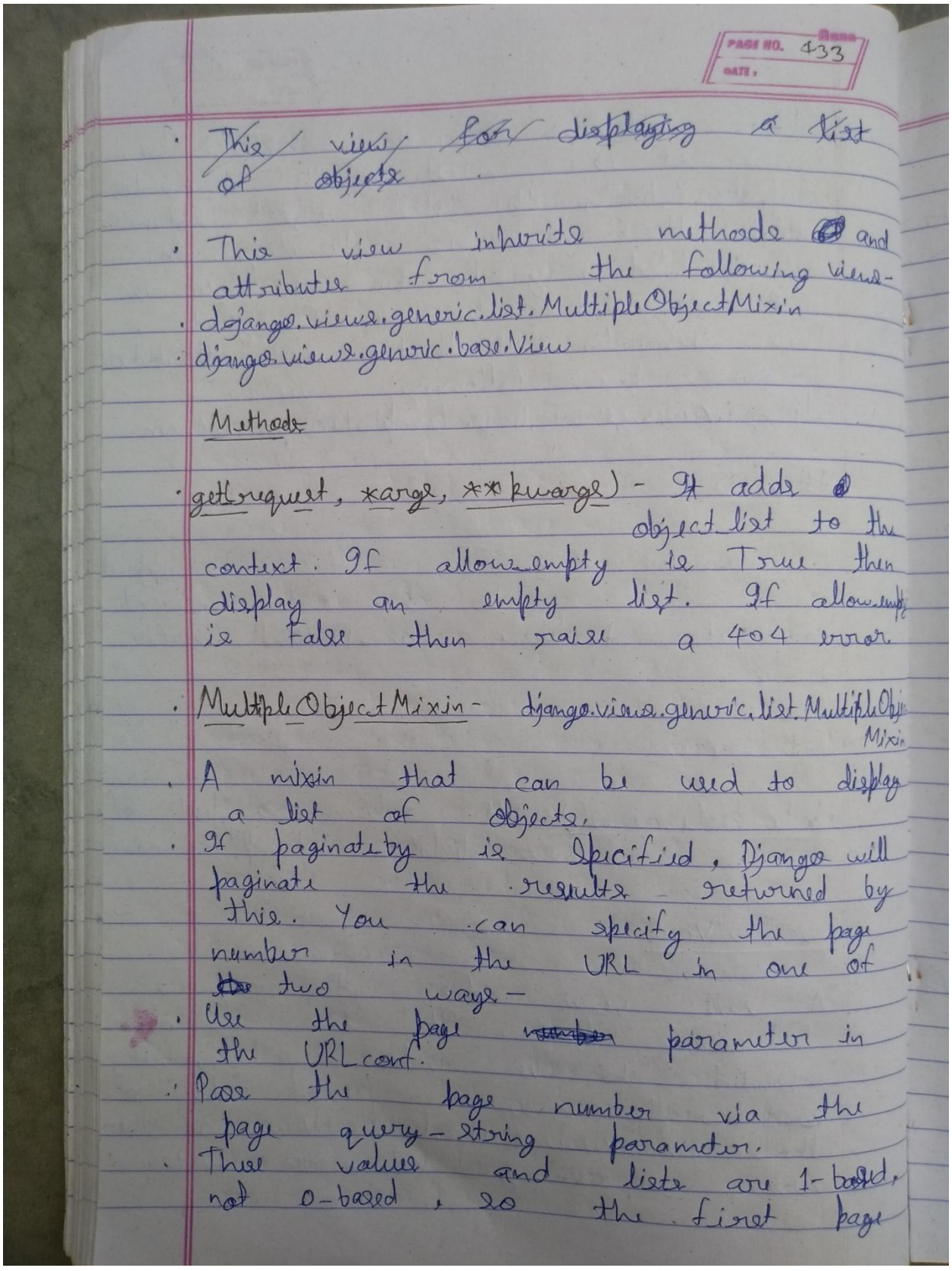
- template_name_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is _list.

- Method-

- get_template_names() - It returns a list of candidate template names.

- BaseListView-

A base view for displaying a list of objects. It is not intended to be used directly, but rather as a parent class of the django.views.generic.list.ListView or other views representing lists of objects.

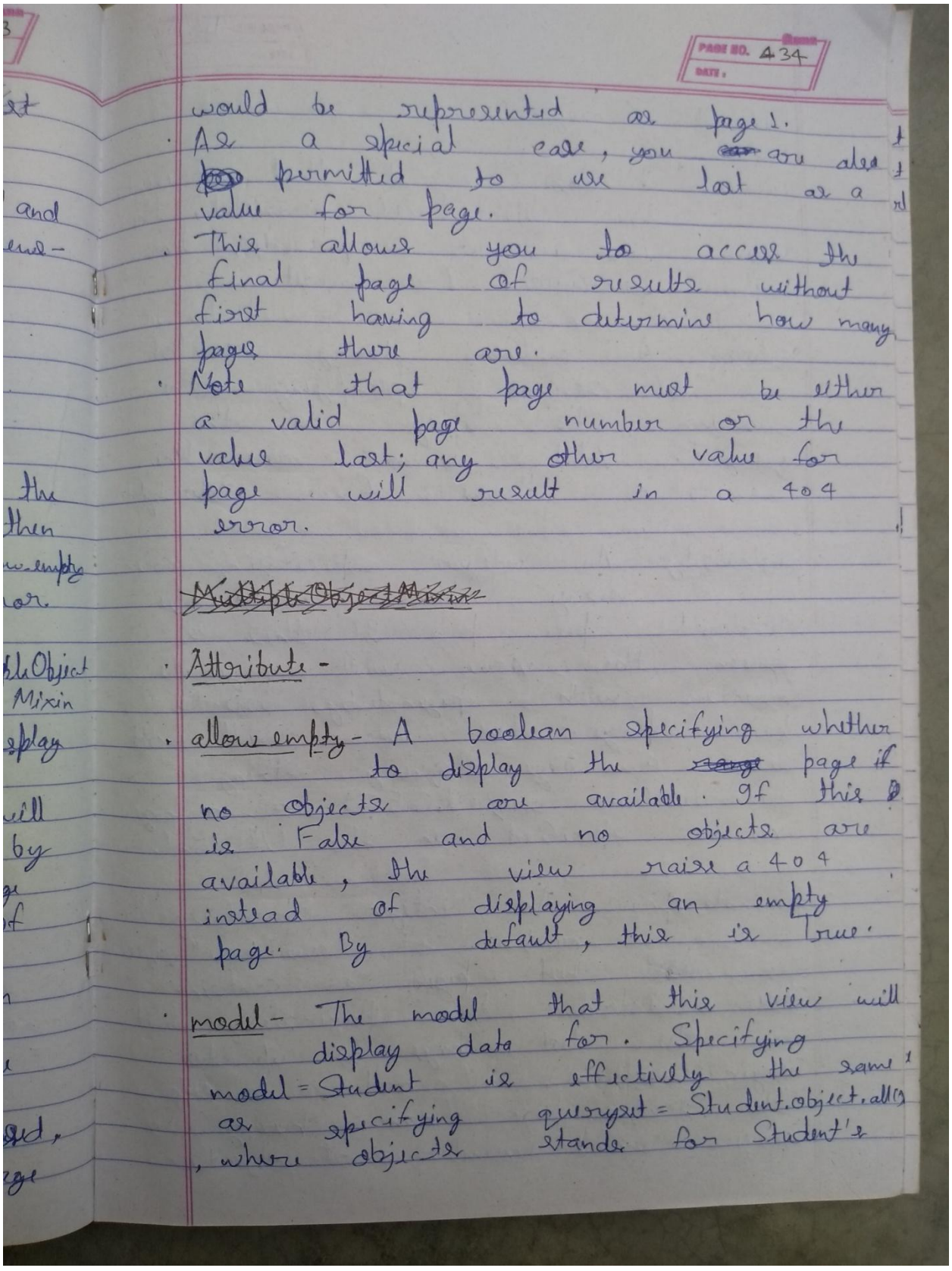


PAGE NO. 433
DATE :

- This view for displaying a list of objects.
- This view inherits methods and attributes from the following views-
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.base.View

Methods

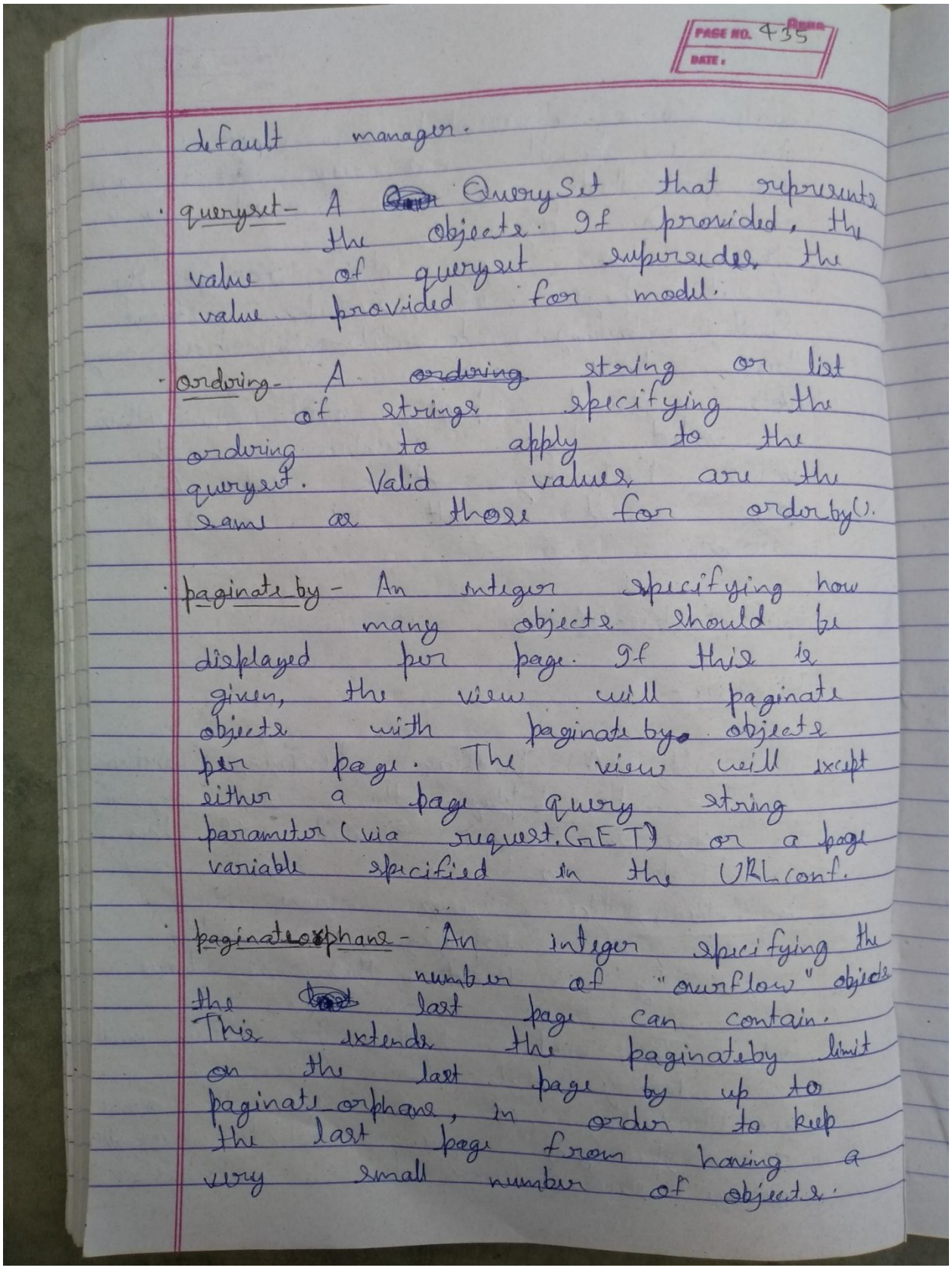
- get(request, *args, **kwargs) - It adds object_list to the context. If allow_empty is True then display an empty list. If allow_empty is false then raise a 404 error.
- MultipleObjectMixin - django.views.generic.list.MultipleObjectMixin
- A mixin that can be used to display a list of objects.
- If paginate_by is specified, Django will paginate the results returned by this. You can specify the page number in the URL in one of two ways -
- Use the page parameter in the URL conf.
- Pass the page number via the page query-string parameter.
- These values and lists are 1-based, not 0-based, so the first page



would be represented as page 1.
- As a special case, you are also permitted to use last as a value for page.
- This allows you to access the final page of results without first having to determine how many pages there are.
- Note that page must be either a valid page number or the value last; any other value for page will result in a 404 error.

~~Multiple Object Mixin~~

- Attributes -
- allow_empty - A boolean specifying whether to display the page if no objects are available. If this is False and no objects are available, the view raise a 404 instead of displaying an empty page. By default, this is True.
- model - The model that this view will display data for. Specifying model = Student is effectively the same as specifying queryset = Student.objects.all(), where objects stands for Student's



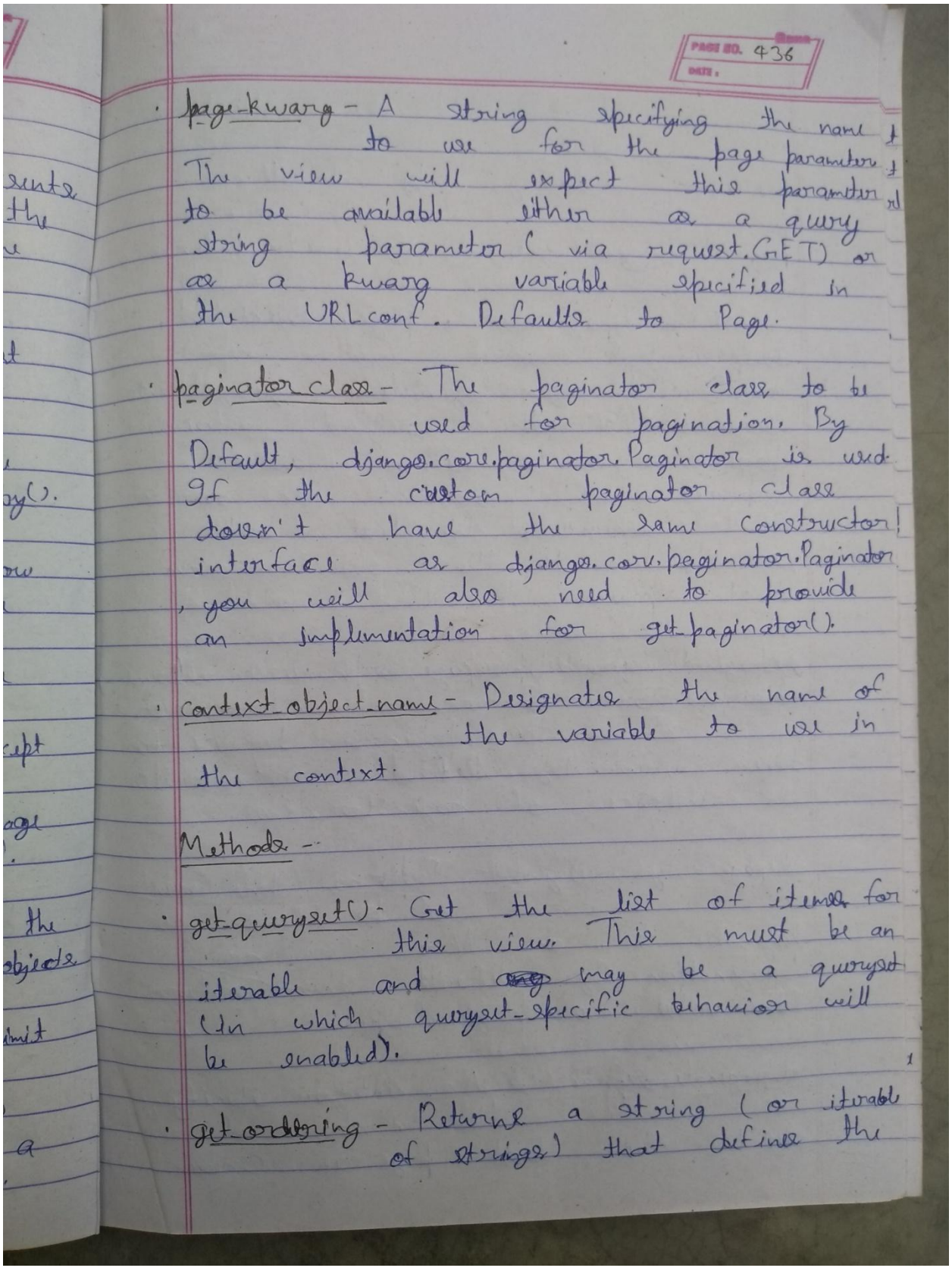
default manager.

- queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

- ordering - A ordering string or list of strings specifying the ordering to apply to the queryset. Valid values are the same as those for order_by().

- paginate_by - An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with paginate_by objects per page. The view will expect either a page query string parameter (via request.GET) or a page variable specified in the URLconf.

- paginate_orphans - An integer specifying the number of "overflow" objects the last page can contain. This extends the paginate_by limit on the last page by up to paginate_orphans, in order to keep the last page from having a very small number of objects.



PAGE NO. 436
DATE :

- page_kwarg - A string specifying the name to use for the page parameter. The view will expect this parameter to be available either as a query string parameter (via request.GET) or as a kwarg variable specified in the URLconf. Defaults to page.

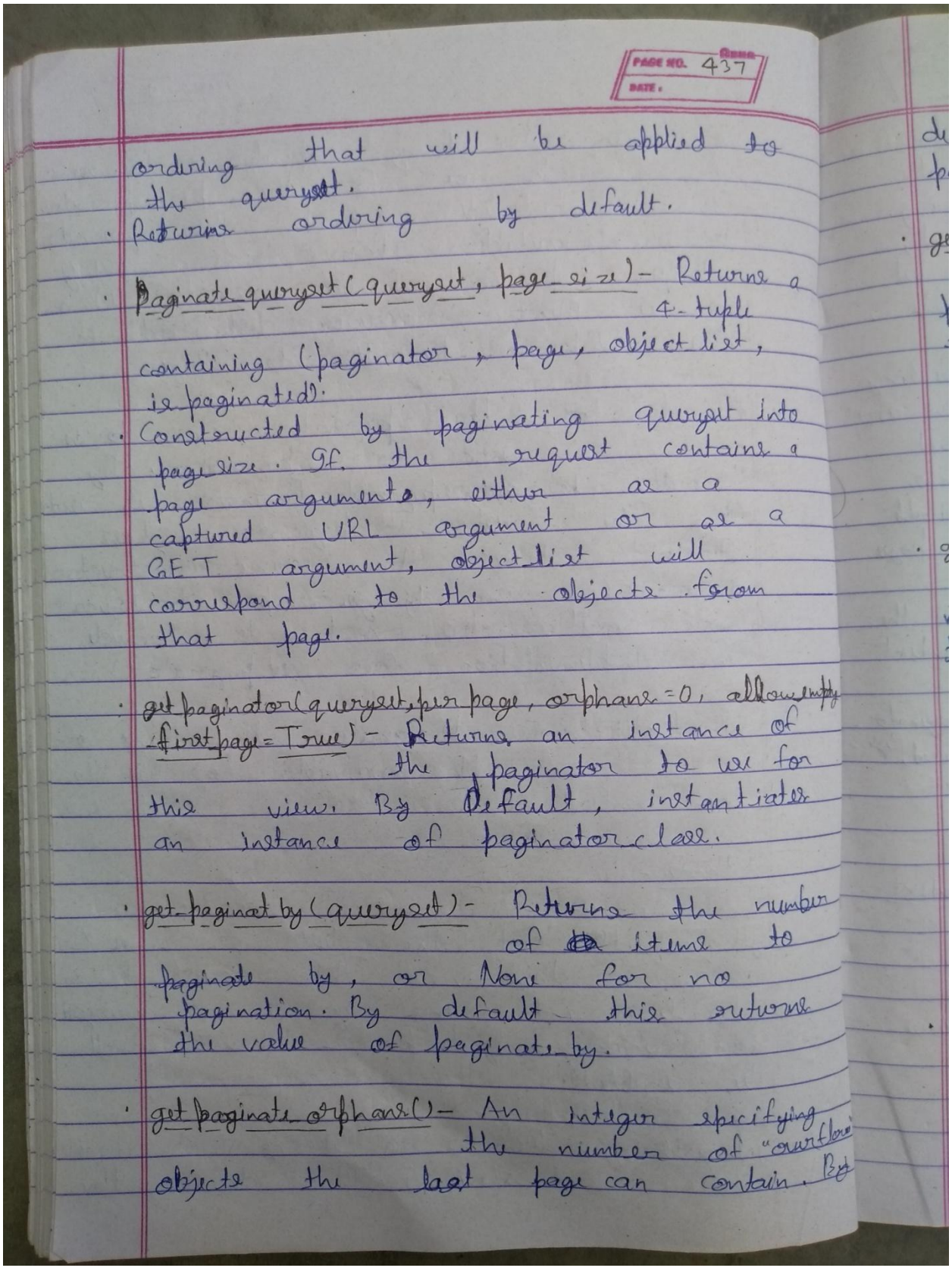
- paginator_class - The paginator class to be used for pagination. By Default, django.core.paginator.Paginator is used. If the custom paginator class doesn't have the same constructor interface as django.core.paginator.Paginator, you will also need to provide an implementation for get_paginator().

- context_object_name - Designates the name of the variable to use in the context.

Methods -

- get_queryset() - Get the list of items for this view. This must be an iterable and may be a queryset (in which queryset-specific behavior will be enabled).

- get_ordering - Returns a string (or iterable of strings) that define the



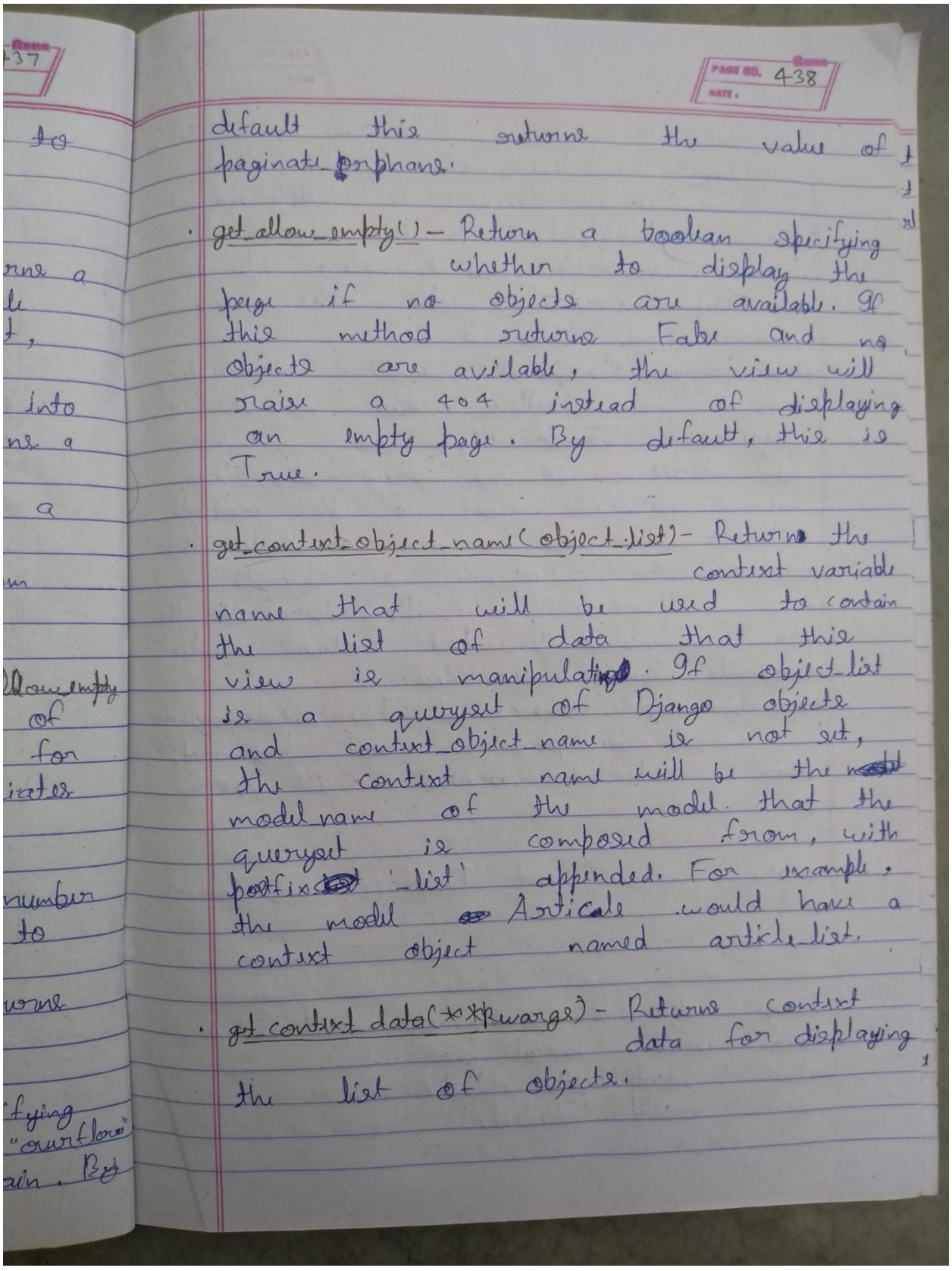
ordering that will be applied to the queryset.
- Returning ordering by default.

- paginate_queryset(queryset, page_size) - Returns a 4-tuple containing (paginator, page, object_list, is_paginated).
- Constructed by paginating queryset into page_size. If the request contains a page argument, either as a captured URL argument or as a GET argument, object_list will correspond to the objects from that page.

- get_paginator(queryset, per_page, orphans=0, allow_empty_first_page=True) - Returns an instance of the paginator to use for this view. By default, instantiates an instance of paginator_class.

- get_paginate_by(queryset) - Returns the number of items to paginate by, or None for no pagination. By default this returns the value of paginate_by.

- get_paginate_orphans() - An integer specifying the number of "orphan" objects the last page can contain. By

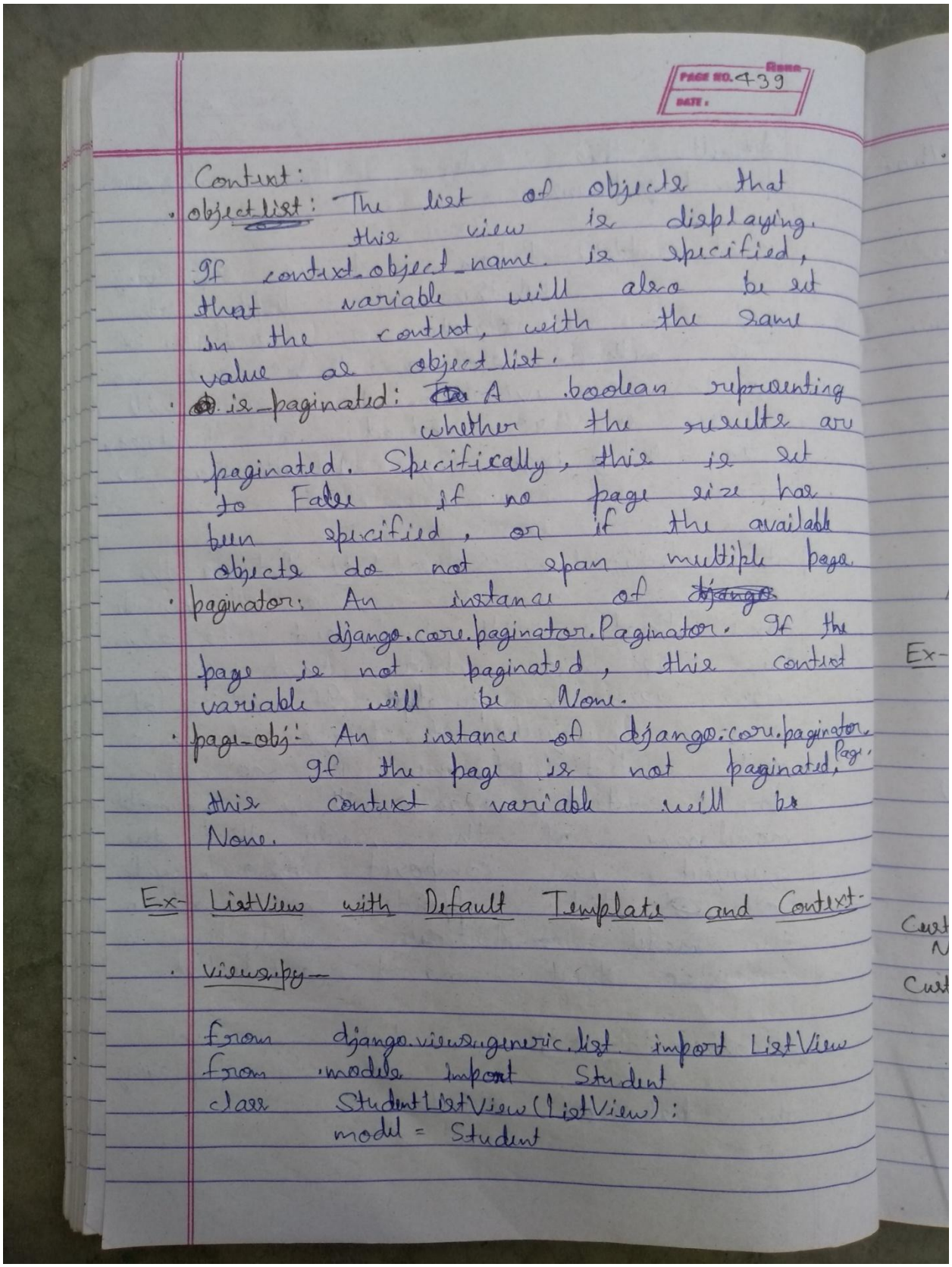


default this returns the value of paginate_orphans.

- get_allow_empty() – Return a boolean specifying whether to display the page if no objects are available. If this method returns False and no objects are avilable, the view will raise a 404 instead of displaying an empty page. By default, this is True.

- get_context_object_name(object_list) – Returns the context variable name that will be used to contain the list of data that this view is manipulating. If object_list is a queryset of Django objects and context_object_name is not set, the context name will be the model_name of the model that the queryset is composed from, with postfix '_list' appended. For example, the model Article would have a context object named article_list.

- get_context_data(**kwargs) – Returns context data for displaying the list of objects.



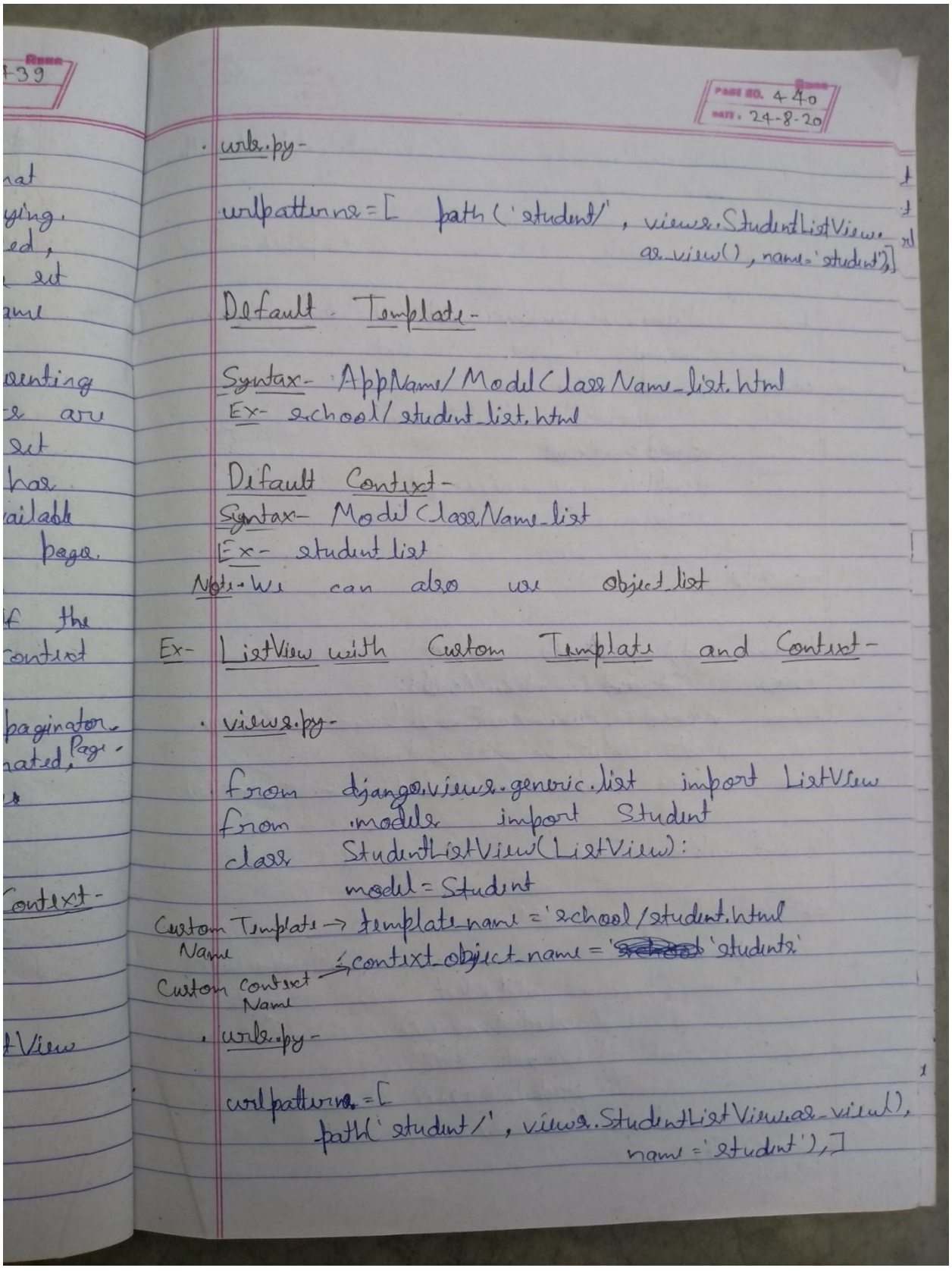
Context:

- object_list: The list of objects that this view is displaying. If context_object_name is specified, that variable will also be set in the context, with the same value as object_list.
- is_paginated: A boolean representing whether the results are paginated. Specifically, this is set to False if no page size has been specified, or if the available objects do not span multiple pages.
- paginator: An instance of ~~django~~ django.core.paginator.Paginator. If the page is not paginated, this context variable will be None.
- page_obj: An instance of django.core.paginator.Page. If the page is not paginated, this context variable will be None.

Ex- ListView with Default Template and Context.

- views.py --

```
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
```



· urls.py -

urlpatterns = [path('student/', views.StudentListView.as_view(), name='student')]

Default Template -

Syntax - AppName/ModelClassName_list.html
Ex - school/student_list.html

Default Context -
Syntax - ModelClassName_list
Ex - student_list
Note - We can also use object_list

Ex - ListView with Custom Template and Context -

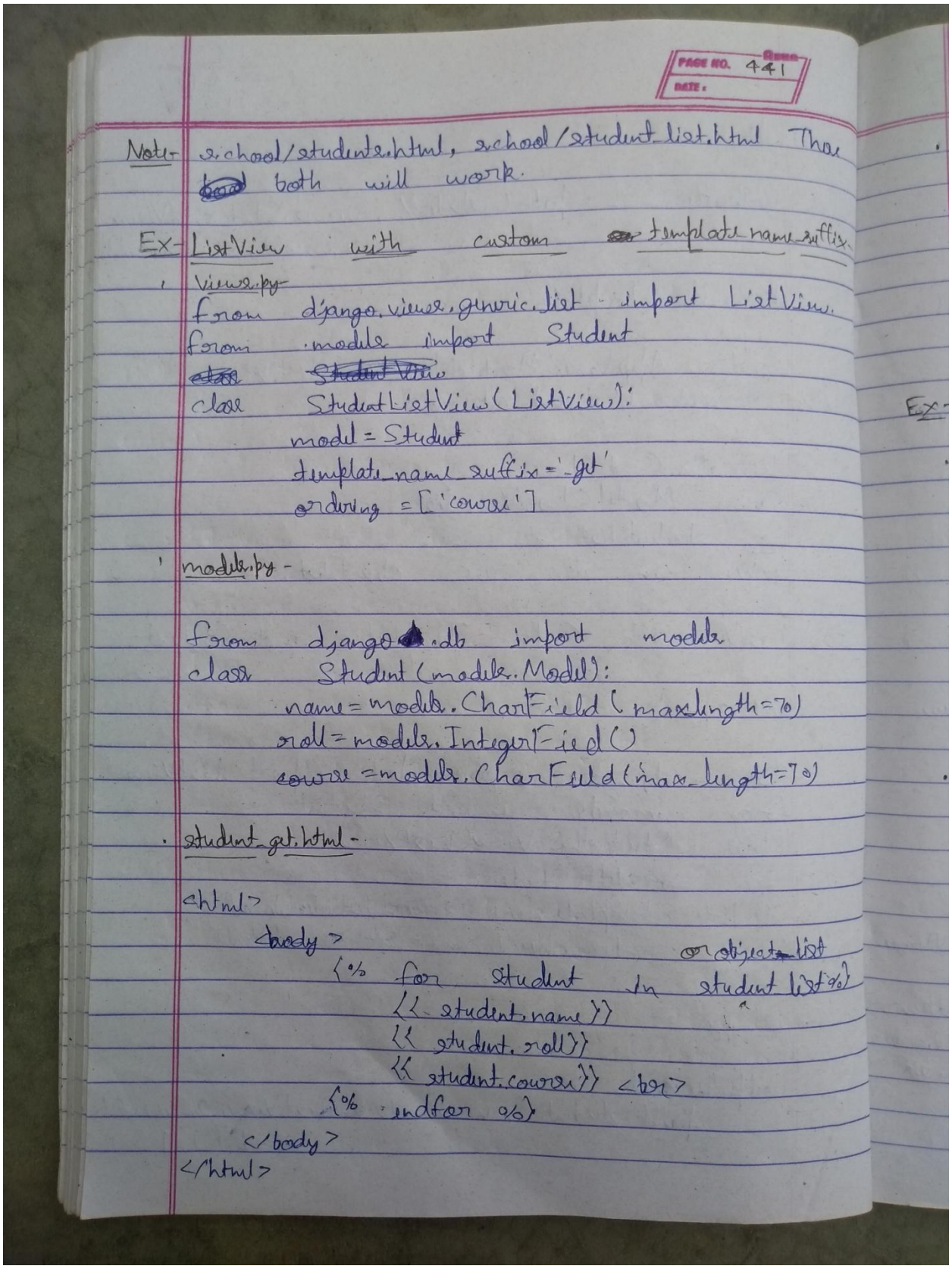
· views.py -

```
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
```

Custom Template Name → template_name = 'school/student.html'
Custom Context Name → context_object_name = 'students'

· urls.py -

```
urlpatterns = [
    path('student/', views.StudentListView.as_view(),
         name='student'),]
```



PAGE NO. 441
DATE :

Note- school/student.html, school/student_list.html Thus both will work.

Ex- ListView with custom template_name_suffix.

views.py -

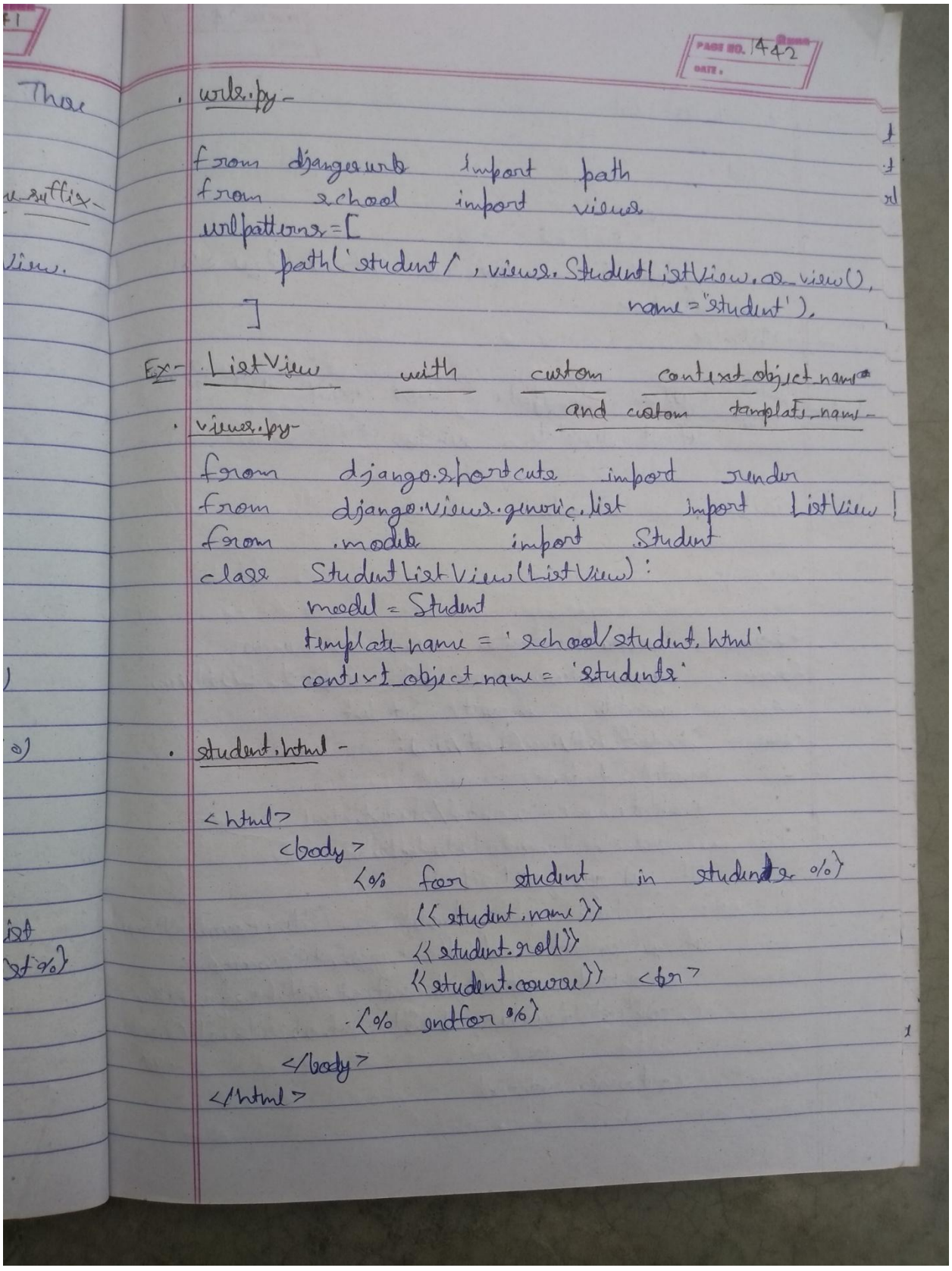
```
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
    template_name_suffix = '_get'
    ordering = ['course']
```

models.py -

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=70)
    roll = models.IntegerField()
    course = models.CharField(max_length=70)
```

student_get.html -

```
<html>
    <body>
        {% for student in student_list %}
            {{ student.name }}
            {{ student.roll }}
            {{ student.course }} <br>
        {% endfor %}
    </body>
</html>
```



PAGE NO. 442
DATE :

. urls.py -

```
from django.urls import path
from school import views
urlpatterns = [
    path('student/', views.StudentListView.as_view(),
                                    name = 'student'),
]
```

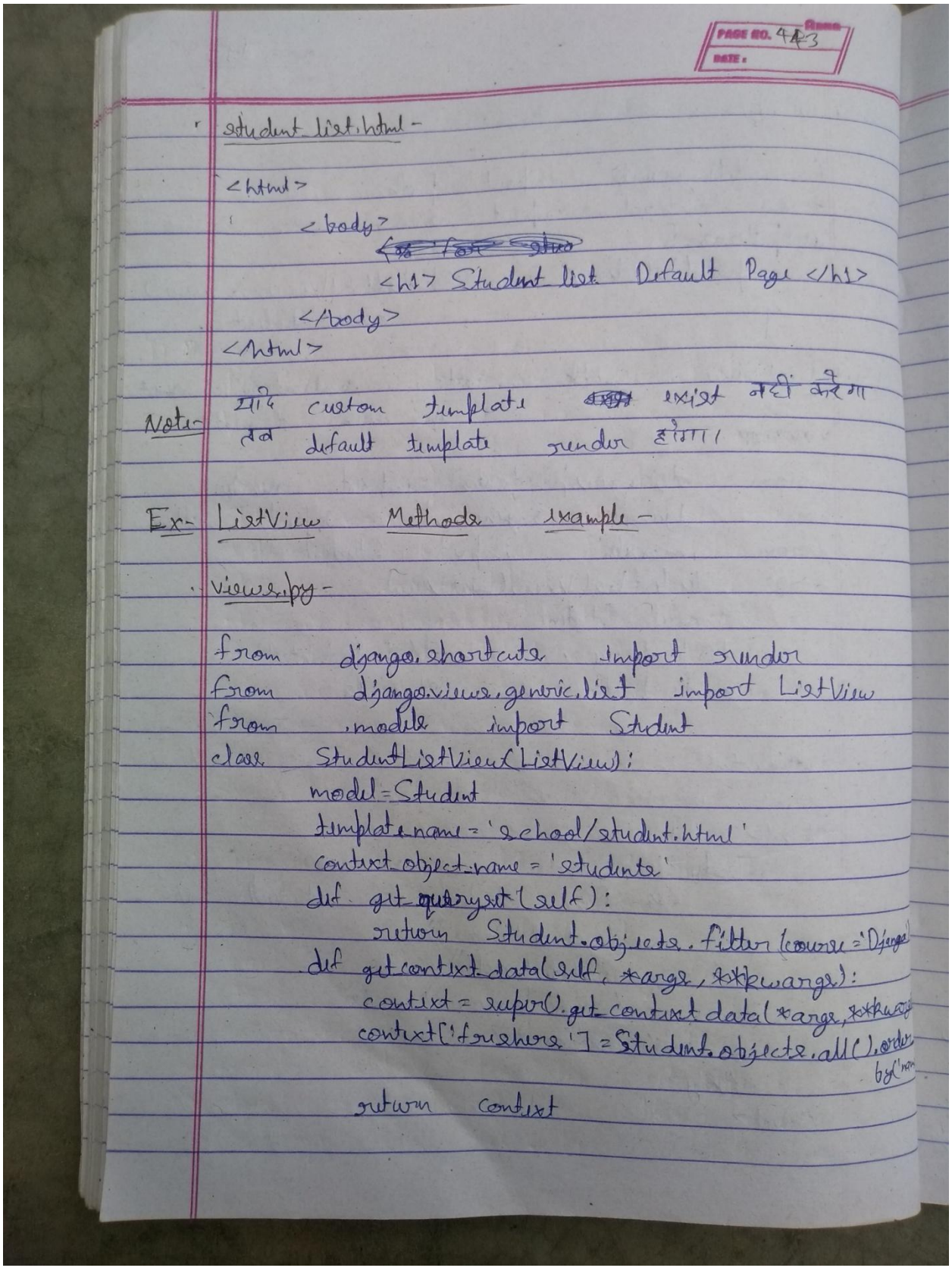
Ex- ListView with custom context_object_name and custom template_name -

. views.py -

```
from django.shortcuts import render
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
    template_name = 'school/student.html'
    context_object_name = 'students'
```

. student.html -

```
<html>
    <body>
        {% for student in students %}
            {{ student.name }}
            {{ student.roll }}
            {{ student.course }} <br>
        {% endfor %}
    </body>
</html>
```



PAGE NO. 443
DATE :

student_list.html -

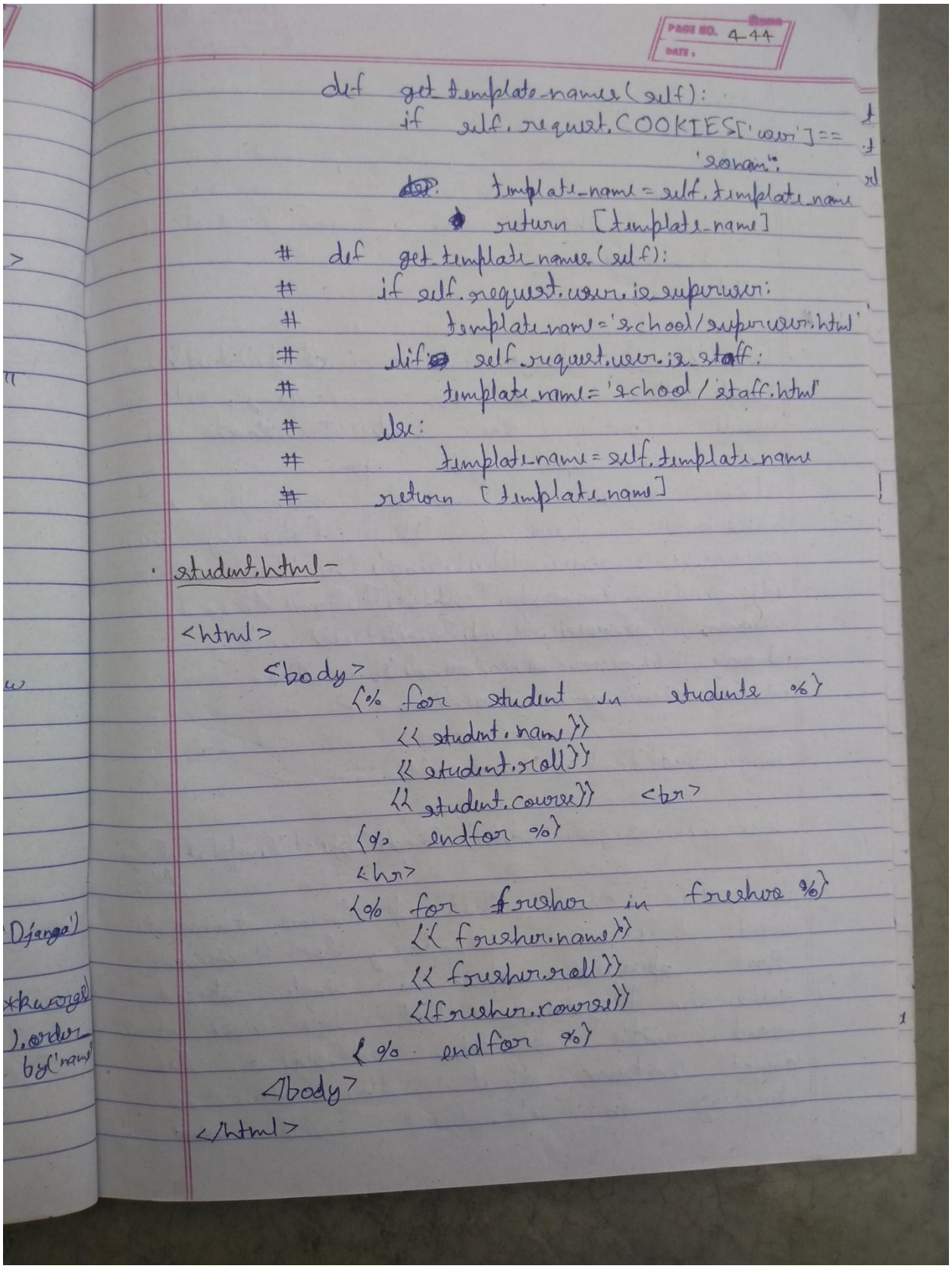
```
<html>
    <body>
        <h1> Student list Default Page </h1>
    </body>
</html>
```

Note- यदि custom template exist नहीं करेगा तो default template render होगा।

Ex- ListView Methods example -

views.py -

```
from django.shortcuts import render
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
    template_name = 'school/student.html'
    context_object_name = 'students'
    def get_queryset(self):
        return Student.objects.filter(course='Django')
    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['freshers'] = Student.objects.all().order_by('name')
        return context
```



```
def get_template_names(self):
    if self.request.COOKIES['user'] ==
                                'sonam':
        template_name = self.template_name
        return [template_name]
# def get_template_names(self):
#     if self.request.user.is_superuser:
#         template_name='school/superuser.html'
#     elif self.request.user.is_staff:
#         template_name='school/staff.html'
#     else:
#         template_name=self.template_name
#     return [template_name]
```

student.html -

```
<html>
    <body>
        {% for student in students %}
            {{ student.name }}
            {{ student.roll }}
            {{ student.course }}  <br>
        {% endfor %}
        <hr>
        {% for fresher in freshers %}
            {{ fresher.name }}
            {{ fresher.roll }}
            {{ fresher.course }}
        {% endfor %}
    </body>
</html>
```

PAGE NO. 445
DATE : 25-8-20

- sonam.html -

```
<html>
    <body>
        <h1> I am Sonam Page </h1>
    </body>
</html>
```

110 —

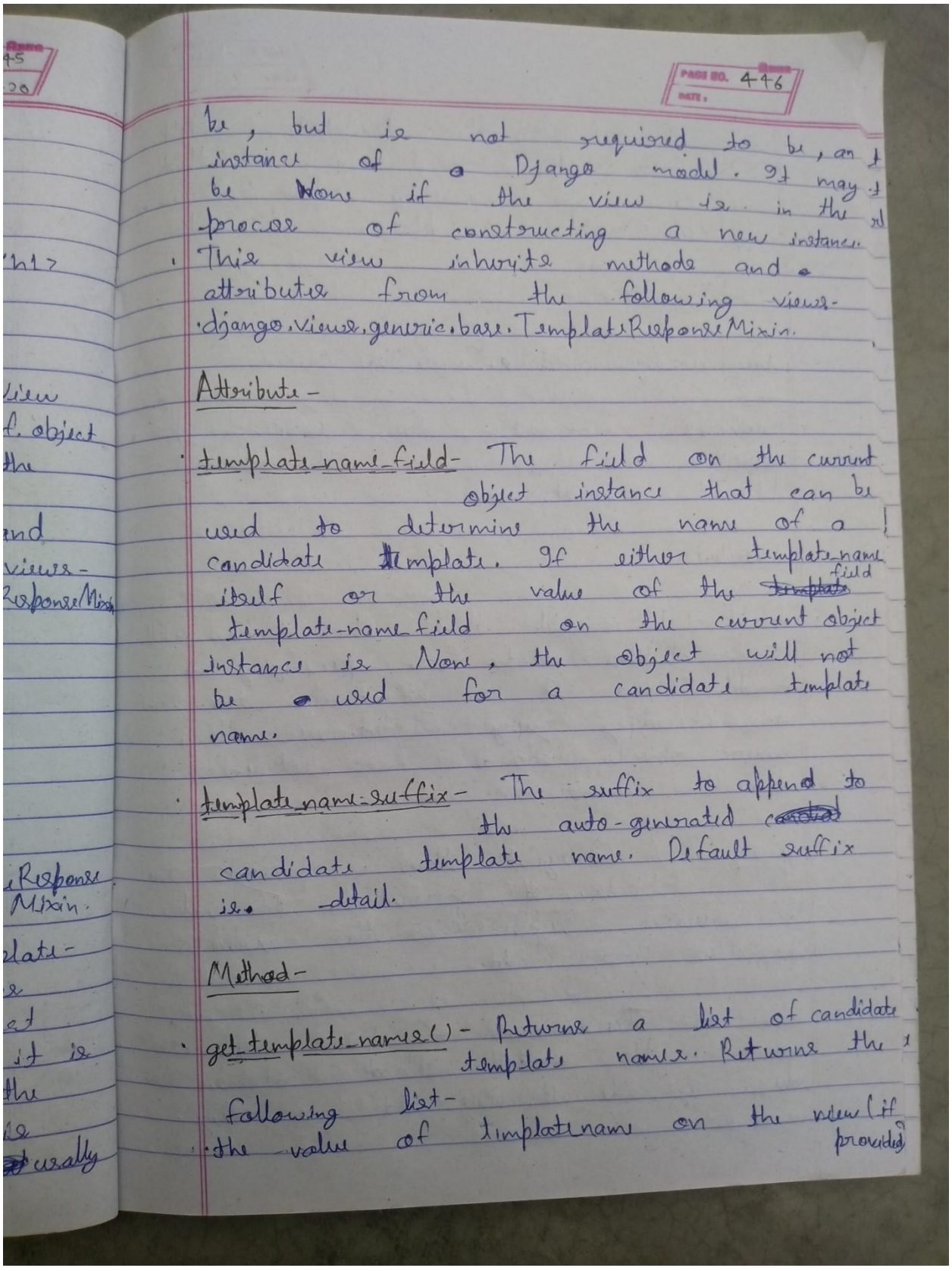
DetailView - django.views.generic.detail.DetailView

- While this view is executing, self.object will contain the object that the view is operating upon.
- This view inherits methods and attributes from the following views -
  - django.views.generic.detail.SingleObjectTemplateResponseMixin
  - django.views.generic.base.TemplateResponseMixin
  - django.views.generic.detail.BaseDetailView
  - django.views.generic.detail.SingleObjectMixin
  - django.views.generic.base.View

- SingleObjectTemplateResponseMixin -

django.views.generic.detail.SingleObjectTemplateResponseMixin

- A mixin class that performs template-based response rendering for views that operate upon a single object instance. Requires that the view it is mixed with provides self.object, the object instance that the view is operating on. self.object will usually



be, but is not required to be, an instance of a Django model. It may be None if the view is in the process of constructing a new instance.

This view inherits methods and attributes from the following views:
django.views.generic.base.TemplateResponseMixin.

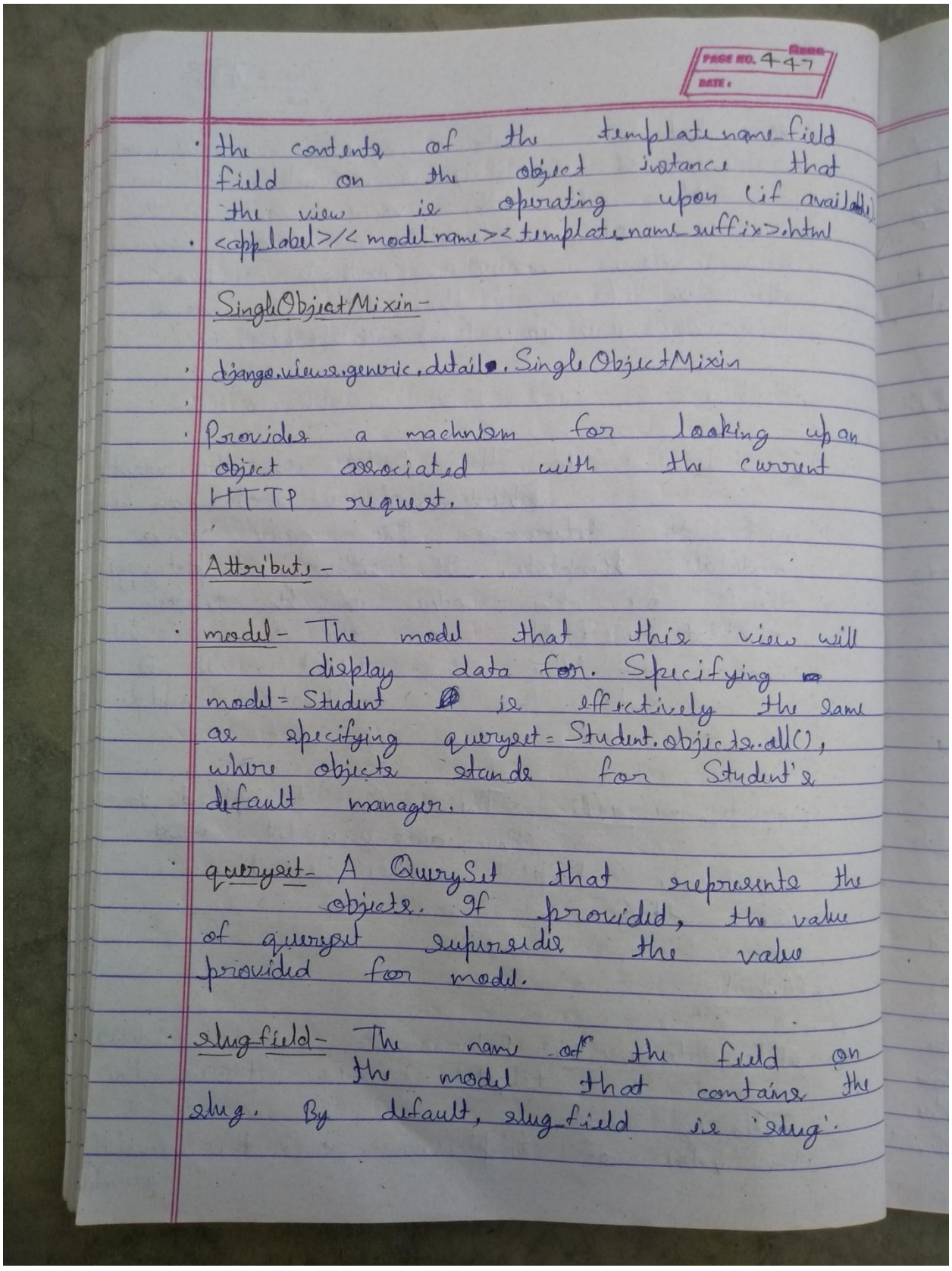
Attribute -

- template_name_field - The field on the current object instance that can be used to determine the name of a candidate template. If either template_name itself or the value of the template_name_field field on the current object instance is None, the object will not be used for a candidate template name.

- template_name_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is _detail.

Method -

- get_template_names() - Returns a list of candidate template names. Returns the following list -
the value of template_name on the view (if provided)



PAGE NO. 447
DATE :

- the contents of the template name field field on the object instance that the view is operating upon (if available).
- <app label>/<model name><template name suffix>.html

SingleObjectMixin -

- django.views.generic.detail.SingleObjectMixin

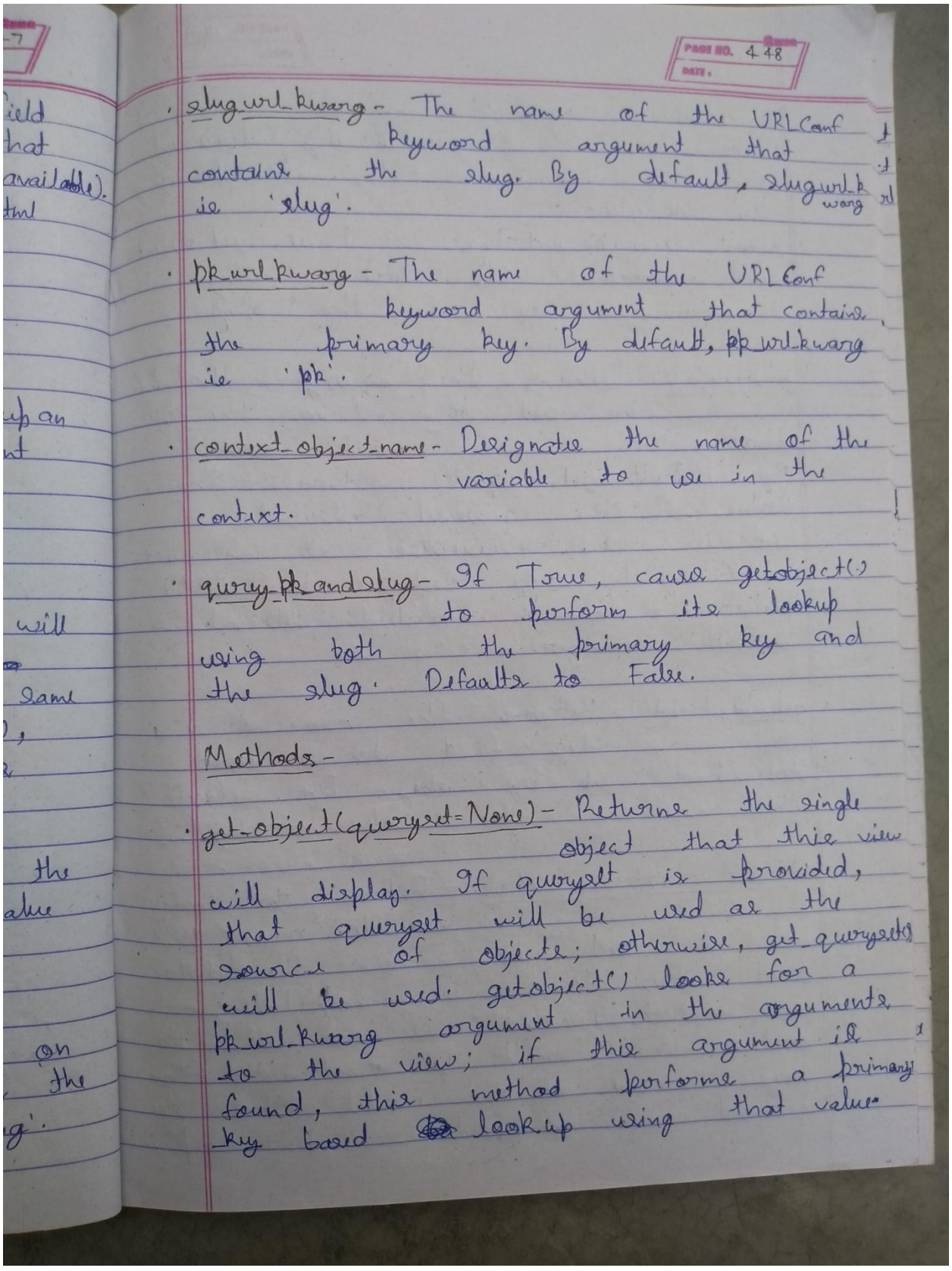
- Provides a machinism for looking up an object associated with the current HTTP request.

Attributes -

- model - The model that this view will display data for. Specifying model = Student is effectively the same as specifying queryset = Student.objects.all(), where objects stands for Student's default manager.

- queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

- slug_field - The name of the field on the model that contains the slug. By default, slug_field is 'slug'.



- slug_url_kwarg - The name of the URLConf keyword argument that contains the slug. By default, slug_url_kwarg is 'slug'.

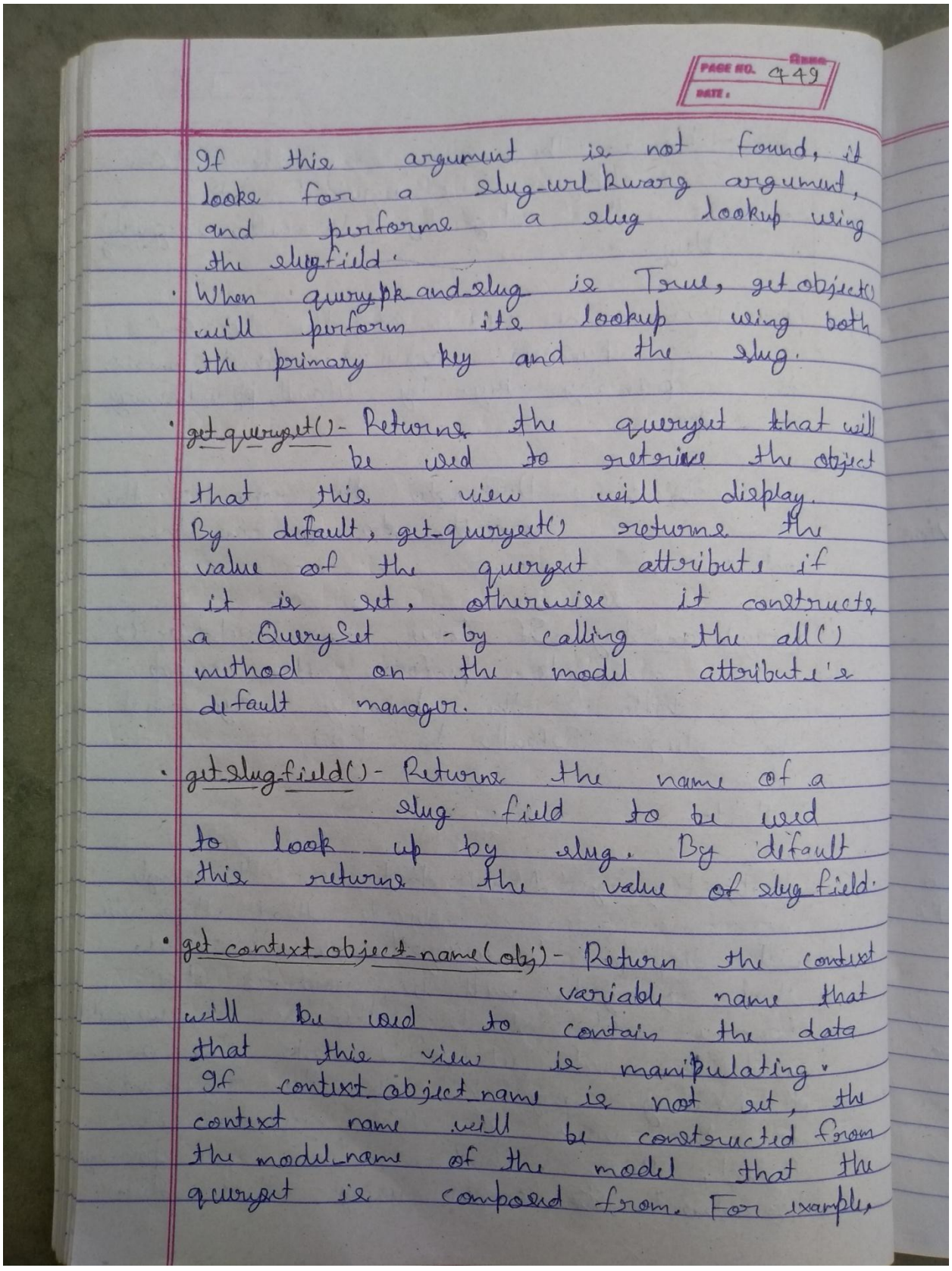
- pk_url_kwarg - The name of the URLConf keyword argument that contains the primary key. By default, pk_url_kwarg is 'pk'.

- context_object_name - Designates the name of the variable to use in the context.

- query_pk_and_slug - If True, causes get_object() to perform its lookup using both the primary key and the slug. Defaults to False.

Methods -

- get_object(queryset=None) - Returns the single object that this view will display. If queryset is provided, that queryset will be used as the source of objects; otherwise, get_queryset() will be used. get_object() looks for a pk_url_kwarg argument in the arguments to the view; if this argument is found, this method performs a primary key based lookup using that value.



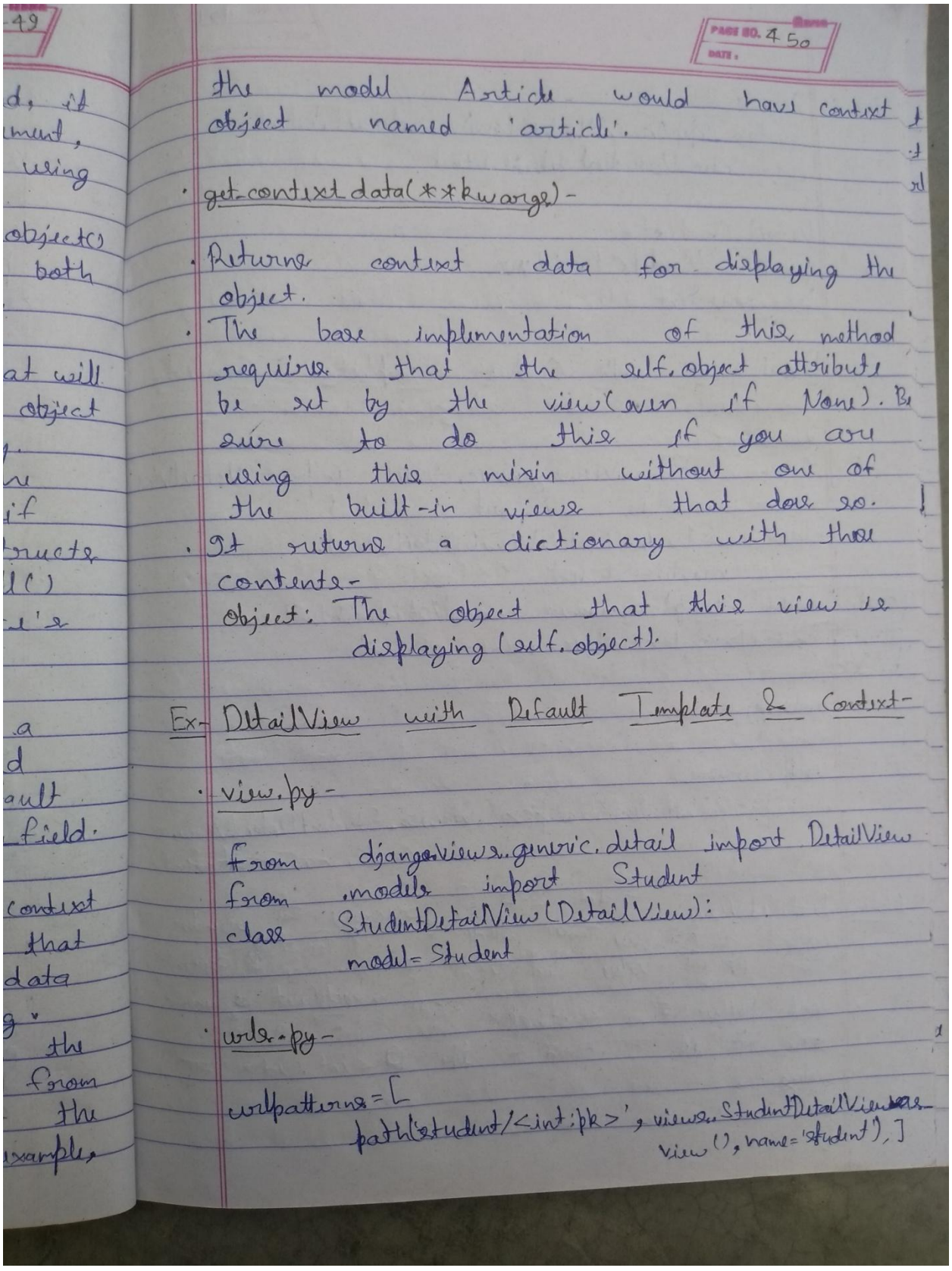
If this argument is not found, it looks for a slug_url_kwarg argument, and performs a slug lookup using the slug_field.

- When query_pk_and_slug is True, get_object() will perform its lookup using both the primary key and the slug.

- get_queryset() - Returns the queryset that will be used to retrieve the object that this view will display. By default, get_queryset() returns the value of the queryset attribute if it is set, otherwise it constructs a QuerySet by calling the all() method on the model attribute's default manager.

- get_slug_field() - Returns the name of a slug field to be used to look up by slug. By default this returns the value of slug_field.

- get_context_object_name(obj) - Return the context variable name that will be used to contain the data that this view is manipulating. If context_object_name is not set, the context name will be constructed from the model_name of the model that the queryset is composed from. For example,



the model Article would have context object named 'article'.

- get_context_data(**kwargs)-
- Returns context data for displaying the object.
- This base implementation of this method requires that the self.object attribute be set by the view (even if None). Be sure to do this if you are using this mixin without one of the built-in views that does so.
- It returns a dictionary with these contents-
  object: The object that this view is displaying (self.object).

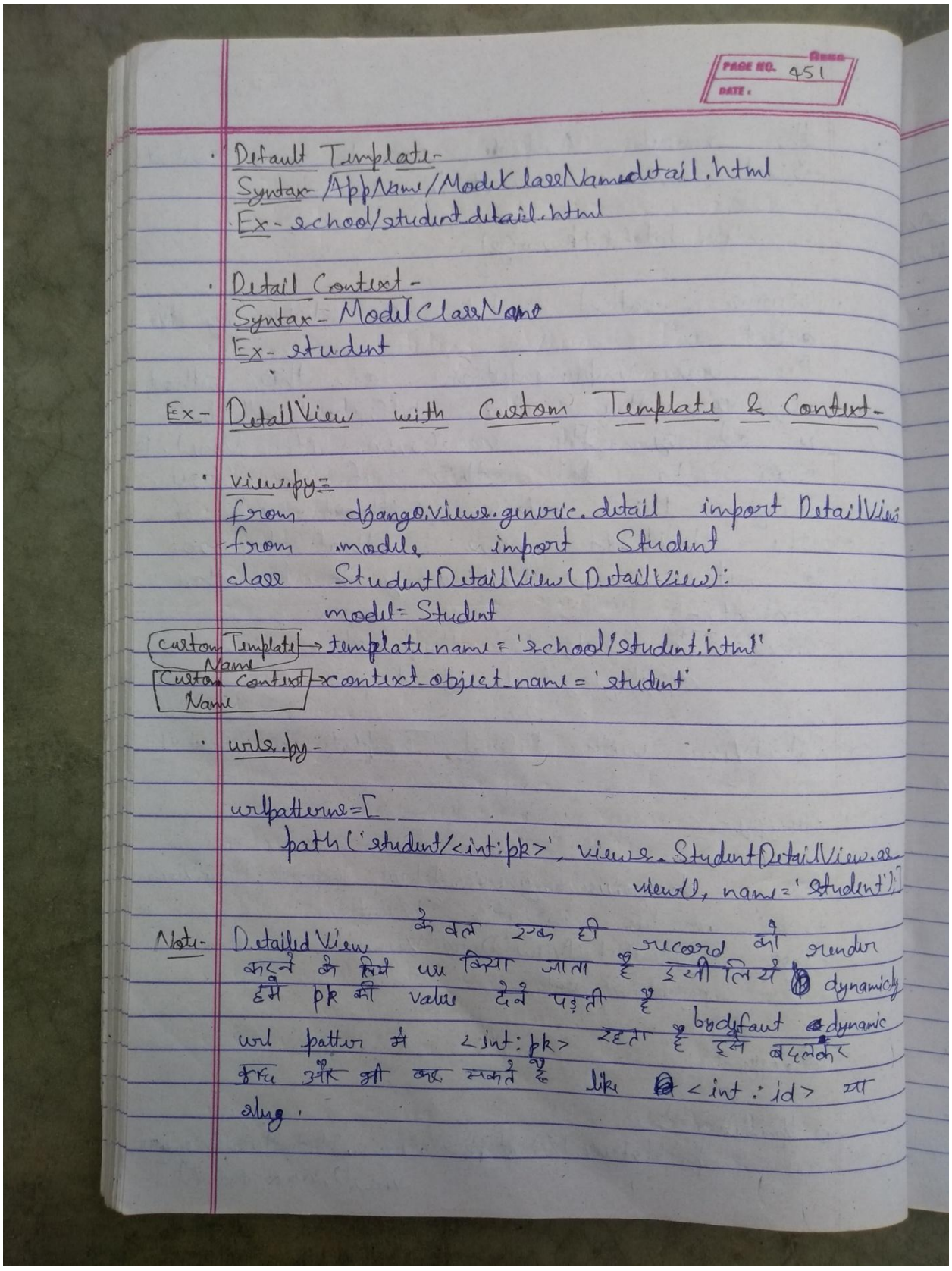
Ex- DetailView with Default Template & Context-

- views.py-

```
from django.views.generic.detail import DetailView
from .models import Student
class StudentDetailView(DetailView):
    model = Student
```

- urls.py-

```
urlpatterns = [
    path('student/<int:pk>', views.StudentDetailView.as_view(), name='student'),]
```



PAGE NO. 451
DATE :

- Default Template-
  Syntax /AppName/ModuleClassName_detail.html
  Ex- school/student_detail.html

- Detail Context-
  Syntax- ModuleClassName
  Ex- student

Ex- DetailView with Custom Template & Context-

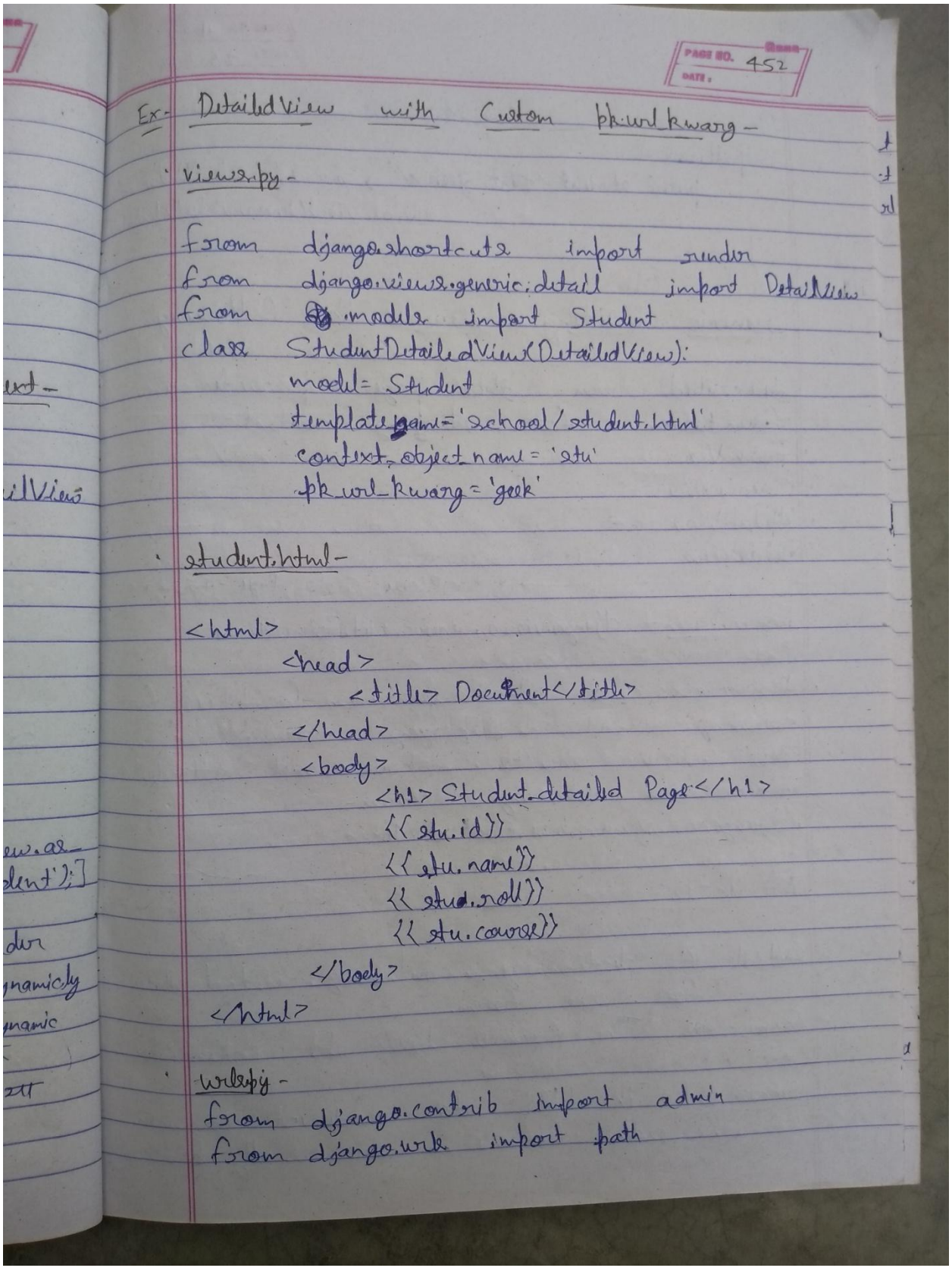
- views.py=

```
from django.views.generic.detail import DetailView
from .models import Student
class StudentDetailView(DetailView):
    model = Student
    template_name = 'school/student.html'      # Custom Template Name
    context_object_name = 'student'            # Custom context Name
```

- urls.py-

```
urlpatterns=[
    path('student/<int:pk>', views.StudentDetailView.as_view(), name='student'),]
```

Note- DetailView केवल एक ही record को render करने के लिए use किया जाता है इसी लिये dynamicly हमें pk की value देनें पड़ती है
url patter में <int:pk> रहता है by default dynamic इसे बदलकर हम और भी कर सकते है like <int:id> या अन्य।



PAGE NO. 452
DATE :

Ex- DetailedView with Custom pk_url_kwarg -

- views.py -

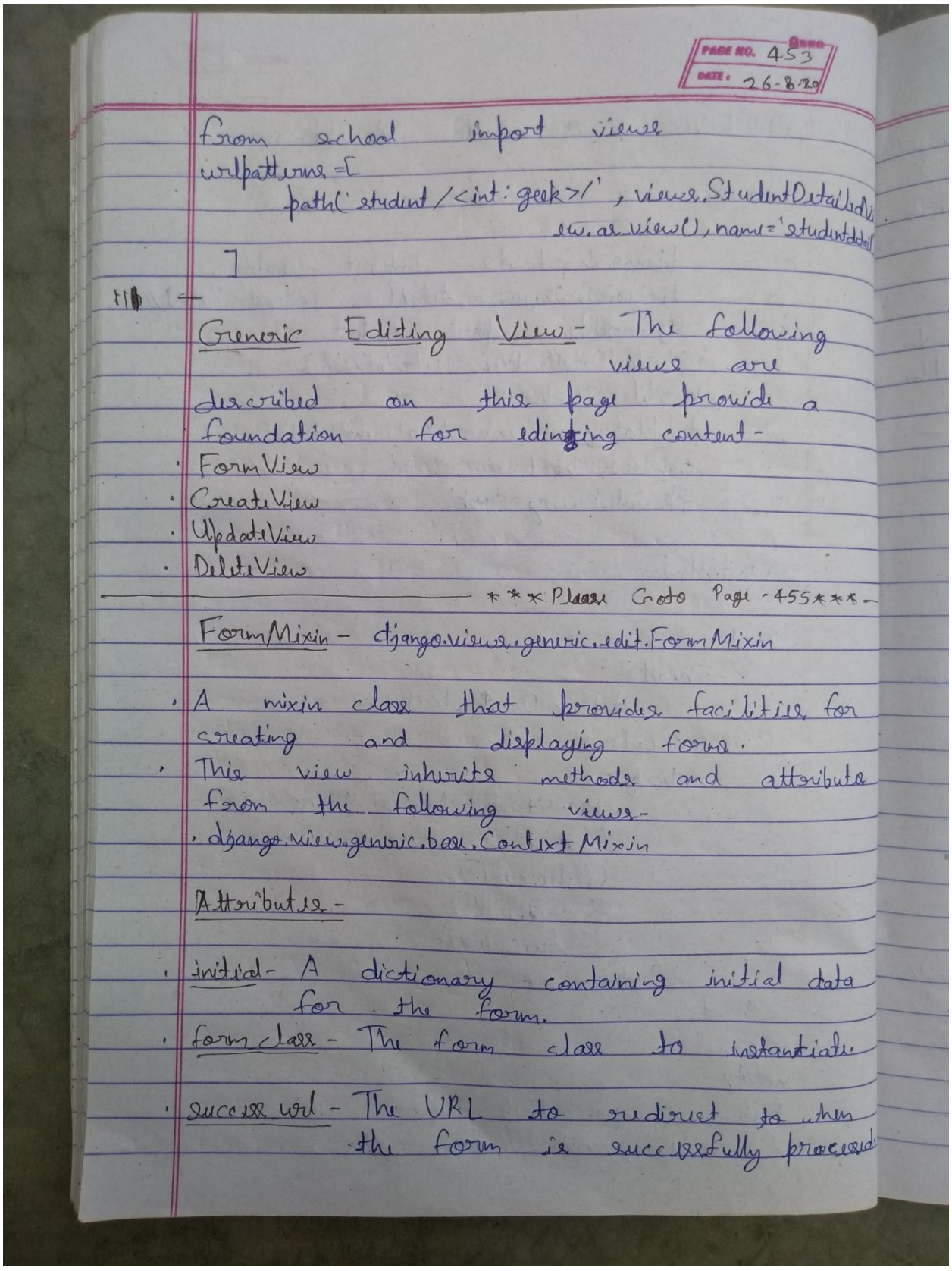
```
from django.shortcuts import render
from django.views.generic.detail import DetailView
from .models import Student
class StudentDetailedView(DetailedView):
    model = Student
    template_name = 'school/student.html'
    context_object_name = 'stu'
    pk_url_kwarg = 'geek'
```

- student.html -

```
<html>
    <head>
        <title> Document</title>
    </head>
    <body>
        <h1> Student_detailed Page</h1>
        {{ stu.id }}
        {{ stu.name }}
        {{ stud.roll }}
        {{ stu.course }}
    </body>
</html>
```

- urls.py -

```
from django.contrib import admin
from django.urls import path
```



PAGE NO. 453
DATE: 26-8-20

```
from school import views
urlpatterns = [
    path('student/<int:geek>/', views.StudentDetailsVi
                                ew.as_view(), name='studentdetail
]
```

Generic Editing View - The following views are described on this page provide a foundation for editing content -

- FormView
- CreateView
- UpdateView
- DeleteView

*** Please Goto Page - 455 ***

FormMixin - django.views.generic.edit.FormMixin

- A mixin class that provides facilities for creating and displaying forms.
- This view inherits methods and attributes from the following views -
  - django.view.generic.base.ContextMixin

Attributes -

- initial - A dictionary containing initial data for the form.
- form_class - The form class to instantiate.
- success_url - The URL to redirect to when the form is successfully proceed