Django REST Framework

GEEKYSHOWS YOUTUBE CHANNEL LEARNING NOTES

Source Code- https://github.com/satyam-seth-learnings/django_rest_learning

Playlist Link- https://youtube.com/playlist?list=PLbGui_ZYuhijTKyrlu-0g5GcP9nUp_HlN

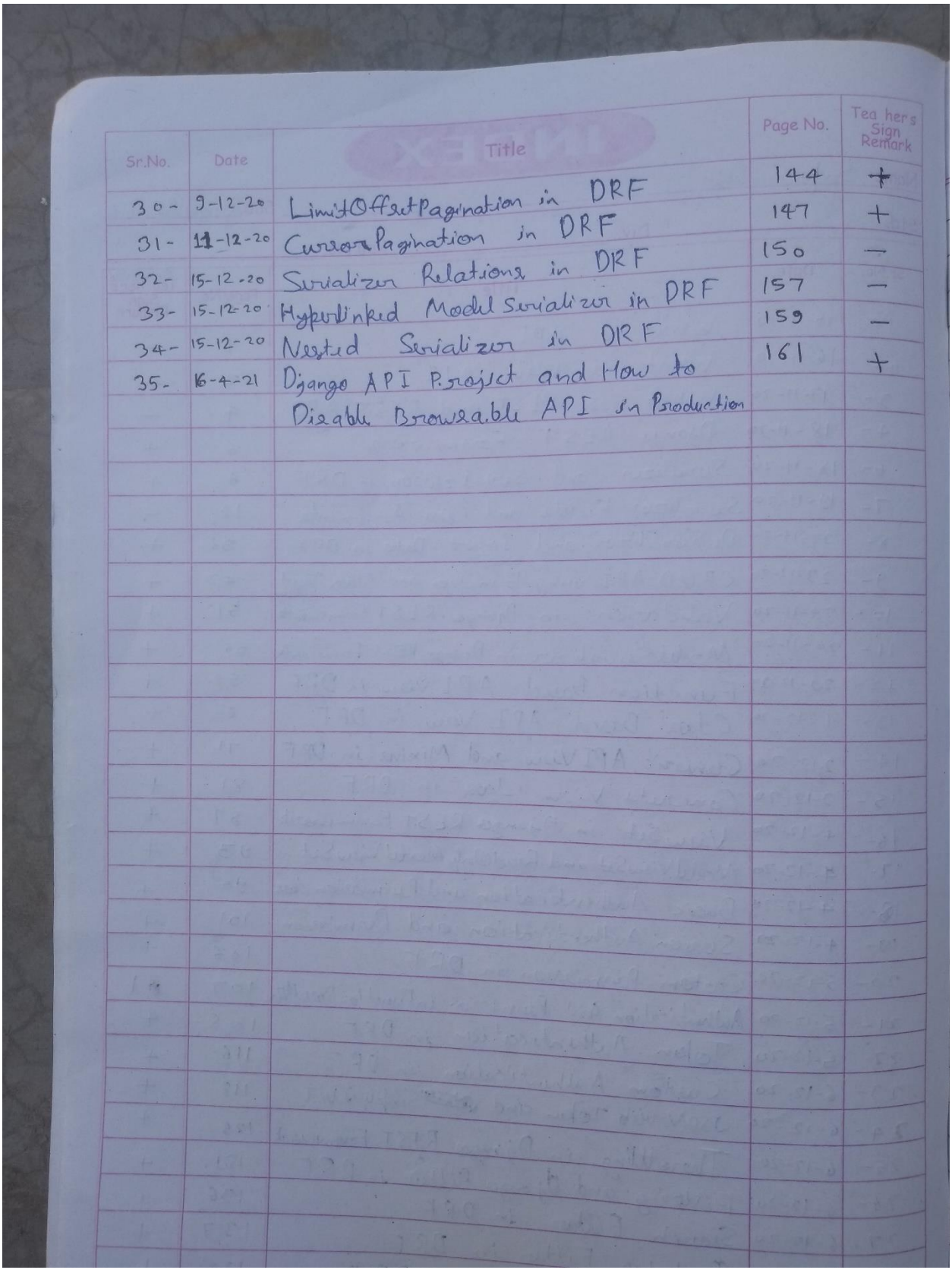
SATYAM SETH

19-09-2021

INDEX

Name Satyam Seth Sub. Django REST framework

Std. Div. Roll No.



Sr.No.	Date	Title	Page No.	Teachers Sign Remark
30-	9-12-20	LimitOffsetPagination in DRF	144	+
31-	11-12-20	CursorPagination in DRF	147	+
32-	15-12-20	Serializer Relations in DRF	150	—
33-	15-12-20	Hyperlinked ModelSerializer in DRF	157	—
34-	15-12-20	Nested Serializer in DRF	159	—
35-	16-4-21	Django API Project and How to Disable Browsable API in Production	161	+

1—

Application Programming Interface (API) —

An API is a software intermediary that allows two or more applications to talk to each other.

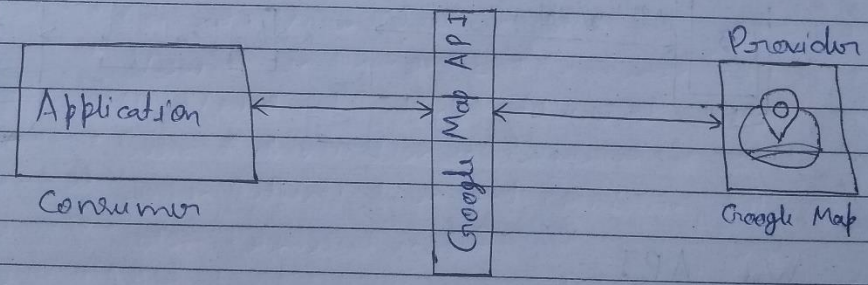
API Types in terms of Release Policies-

1- Private- It can be used within the organization.

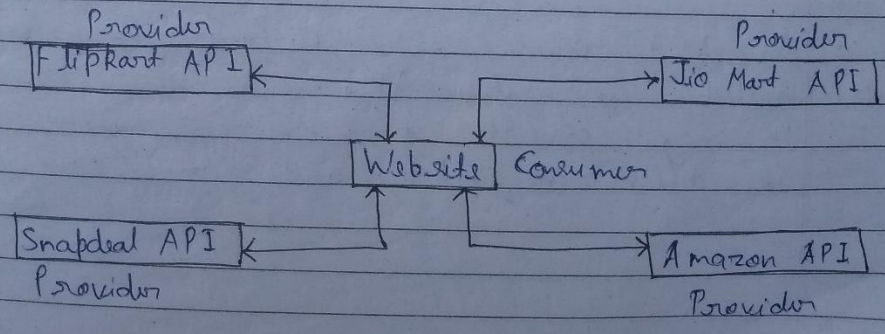
2- Partner- It can be used within Business Partners.

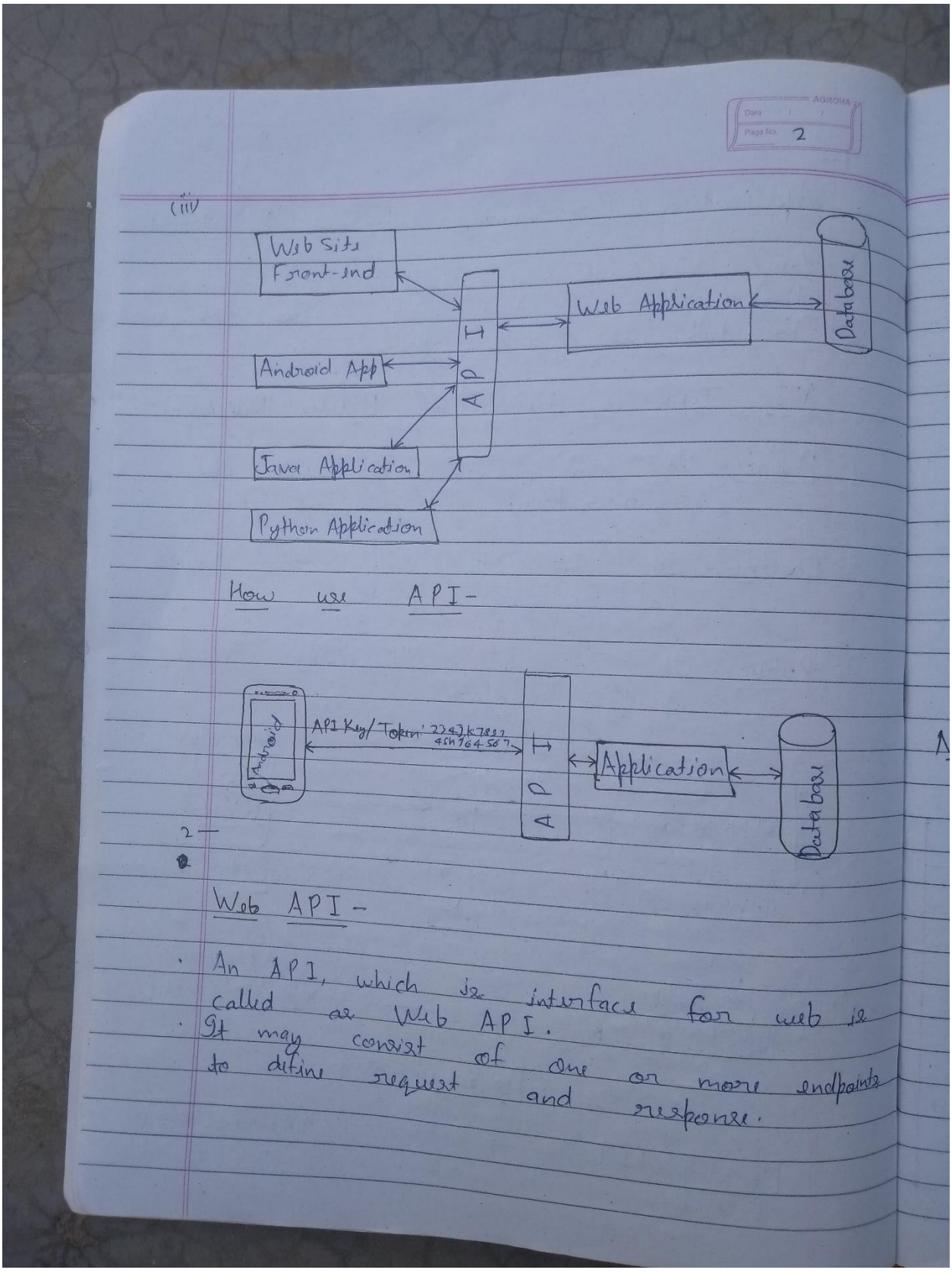
3- Public- It can be used any third party Developers.

Eg- (i)



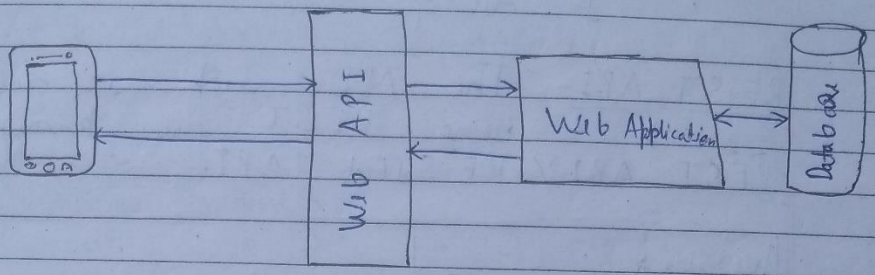
(ii)




AGROHA
Date
Page No. 2
(iii)
Web Site Front-end
Android App
Java Application
Python Application
API
Web Application
Database
How use API-
Android
API Key/ Token: 2249jk7891 45h764567
API
Application
Database
2 —
Web API -
An API, which is interface for web is called as Web API.
It may consist of one or more endpoints to define request and response.

AGROHA
Date 16 11 '20
Page No. 3

How Web API Works-



- Client makes HTTP Request to API.
- API will communicate to Web Application/Database (if needed).
- Web Application / Database provide required data to API.
- API returns Data to Client.

Note- Data in the form of - Json Data / XML Data.

- Register / Sign-up to API.
- API may provide API Key for Authentication purpose -

  API Key / Token : 23423jkhjhgh32434hjgjh4343

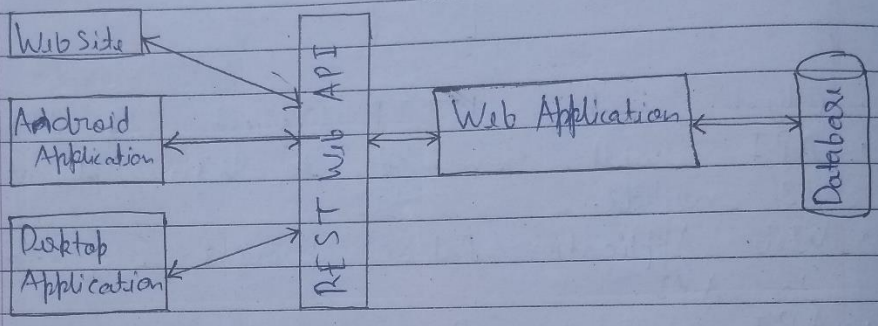
- Whenever you need to communicate with server make Request to API with API Key -
  - http://geekyshows.com/?key=23423jkhjhgh32434hjgjh4343
- If API Key authentication succed, API will provide required Data.

3 —

AGROHA
Date 17/11/20
Page No. 4

REST - It is an architectural guidline to develop Web API.

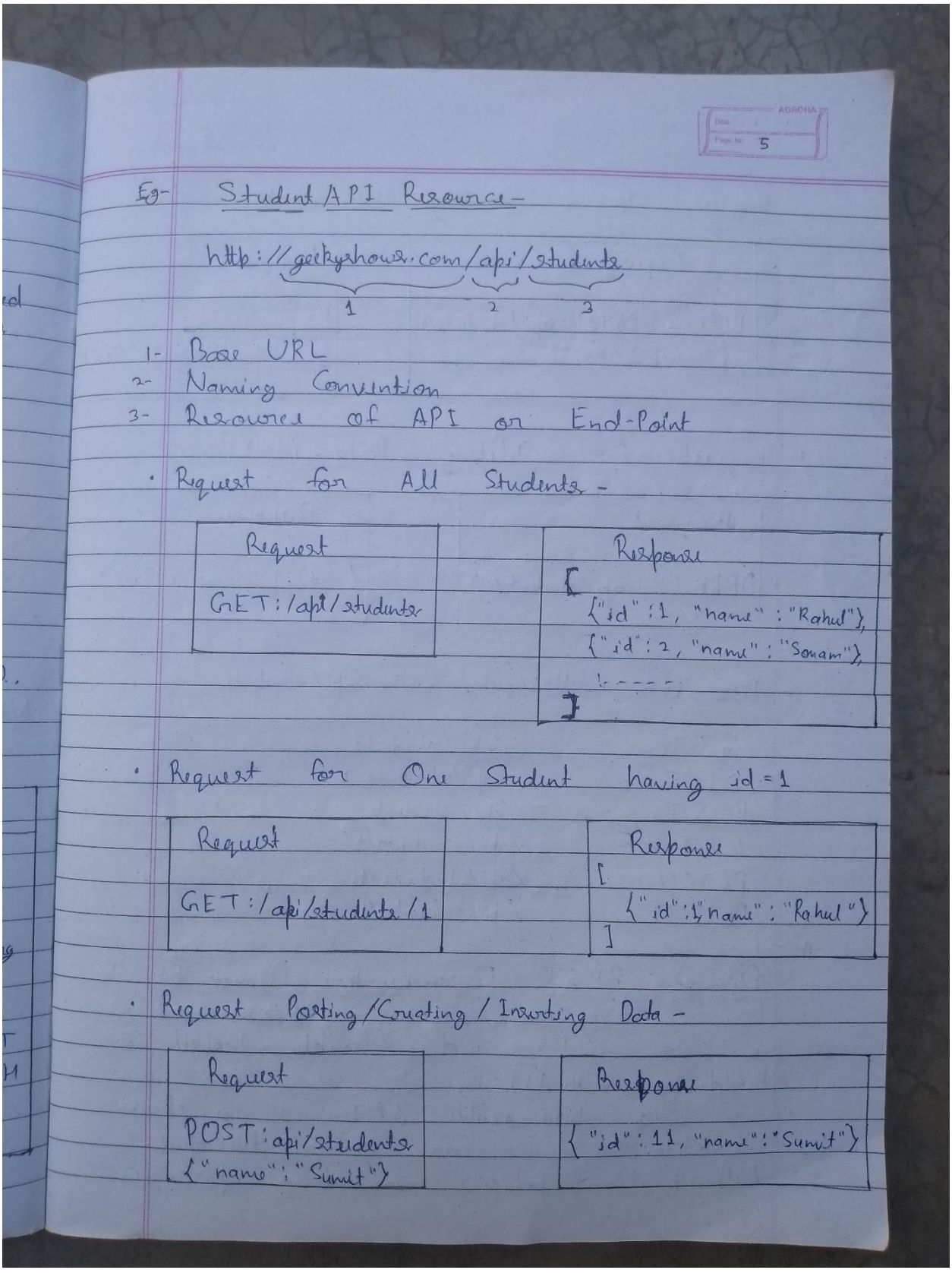
REST API - The API which is developed using REST is known as REST API / RESTful API.



Note - It work same as Web API work (P.T.O.).

CRUD Operations -

Operation	HTTP Methods	Description
Create	POST	Creating / Posting / Inserting Data
Read	GET	Reading / Getting / Retriving Data
Update	PUT, PATCH	Updating Data - · Complete Update - PUT · Partial Update - PATCH
Delete	DELETE	Deleting Data



Eg- Student API Resource -

http://geekyshows.com/api/students
1 2 3

1- Base URL
2- Naming Convention
3- Resource of API or End-Point

• Request for All Students -

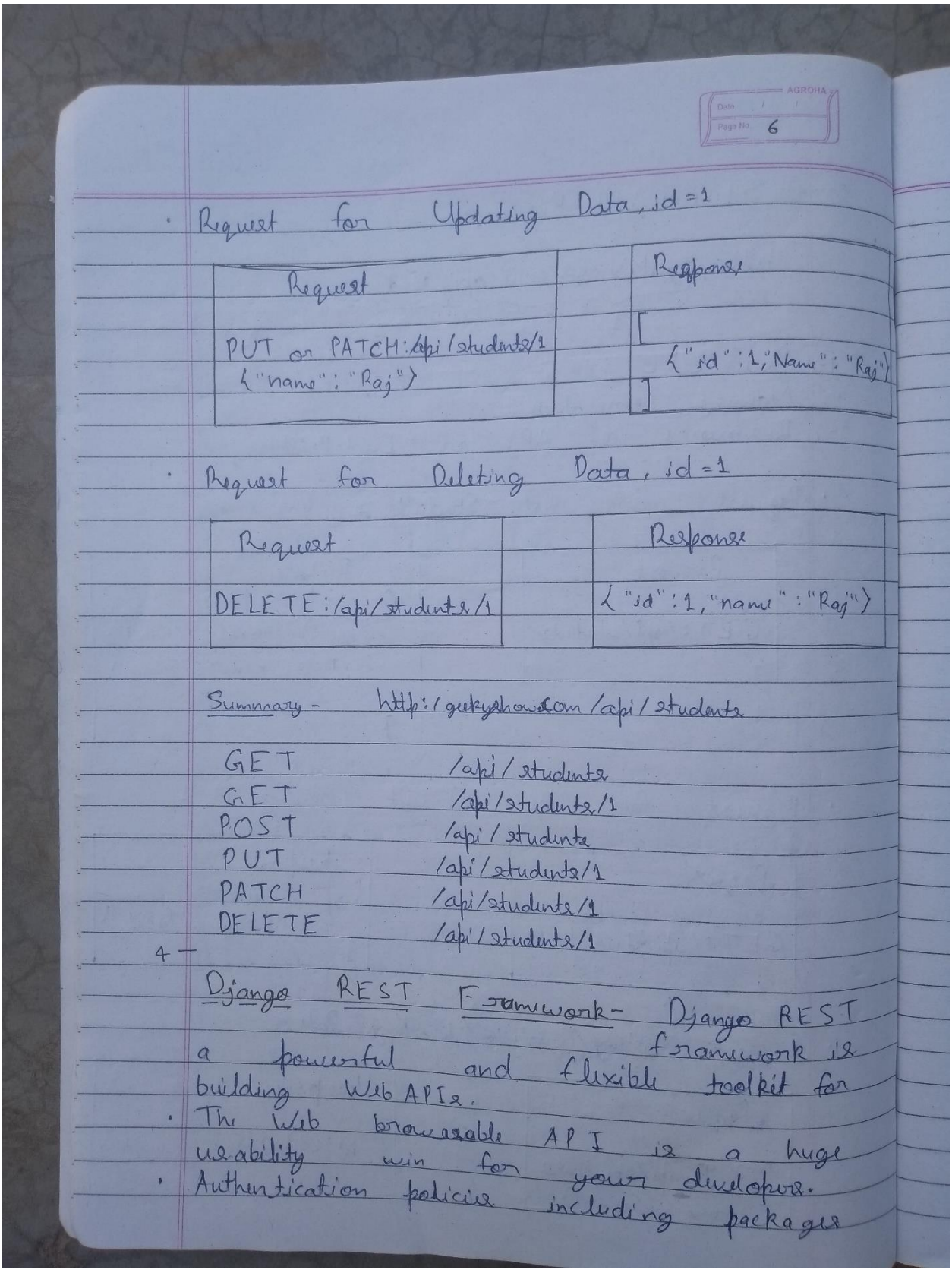
Request	Response
GET: /api/students	[{"id":1, "name":"Rahul"}, {"id":2, "name":"Sonam"}, ...]

• Request for One Student having id = 1

Request	Response
GET: /api/students/1	[{"id":1, "name":"Rahul"}]

• Request Posting / Creating / Inserting Data -

Request	Response
POST: api/students {"name":"Sumit"}	{"id":11, "name":"Sumit"}



AGROHA
Date / /
Page No. 6

- Request for Updating Data, id=1

Request	Response
PUT or PATCH: /api/students/1 {"name": "Raj"}	[{"id": 1, "Name": "Raj"}]

- Request for Deleting Data, id=1

Request	Response
DELETE: /api/students/1	{"id": 1, "name": "Raj"}

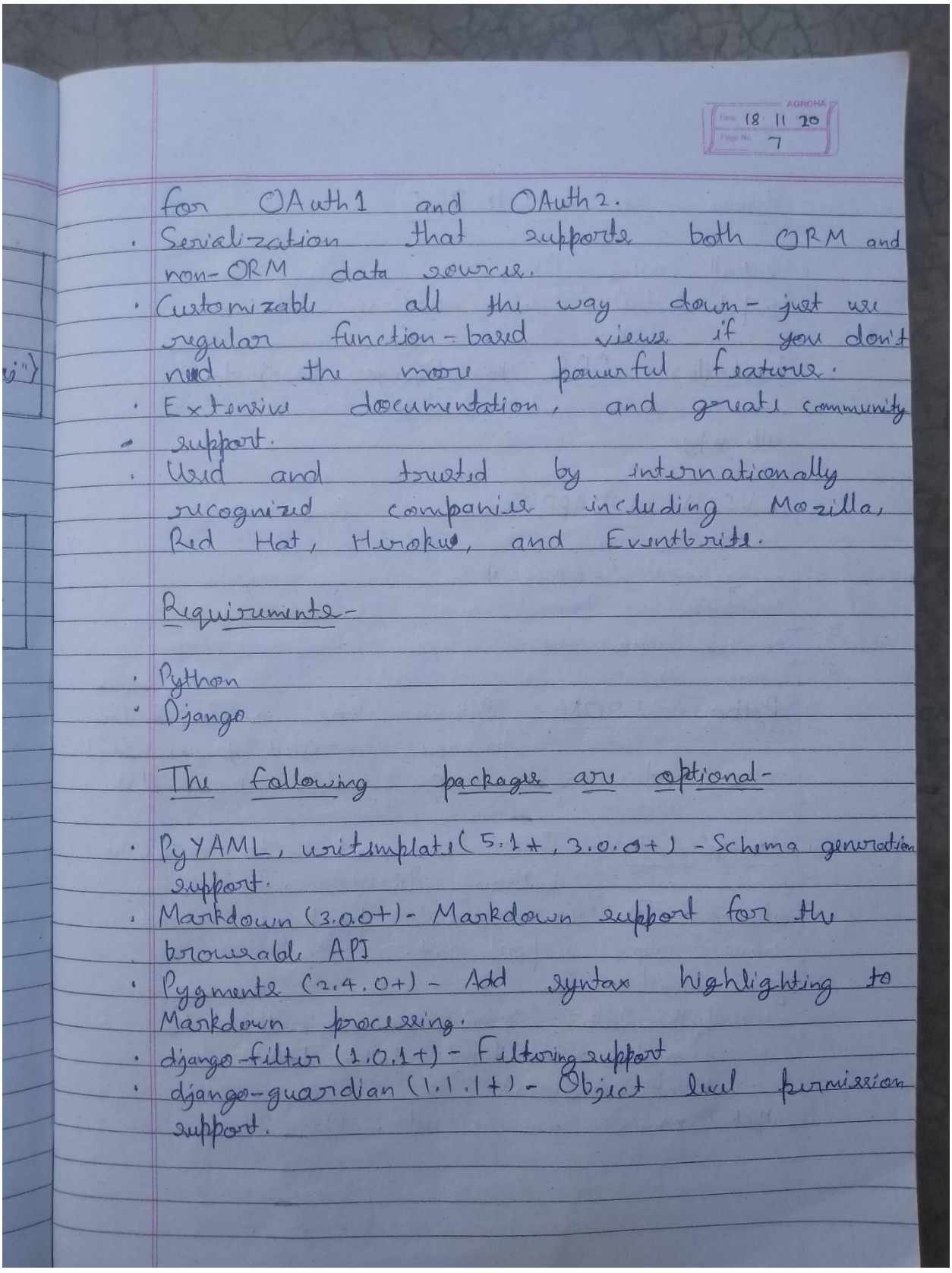
Summary - http://geekyshows.com/api/students

GET	/api/students
GET	/api/students/1
POST	/api/students
PUT	/api/students/1
PATCH	/api/students/1
DELETE	/api/students/1

4 -

Django REST Framework - Django REST framework is a powerful and flexible toolkit for building Web APIs.

- The Web browsable API is a huge usability win for your developers.
- Authentication policies including packages



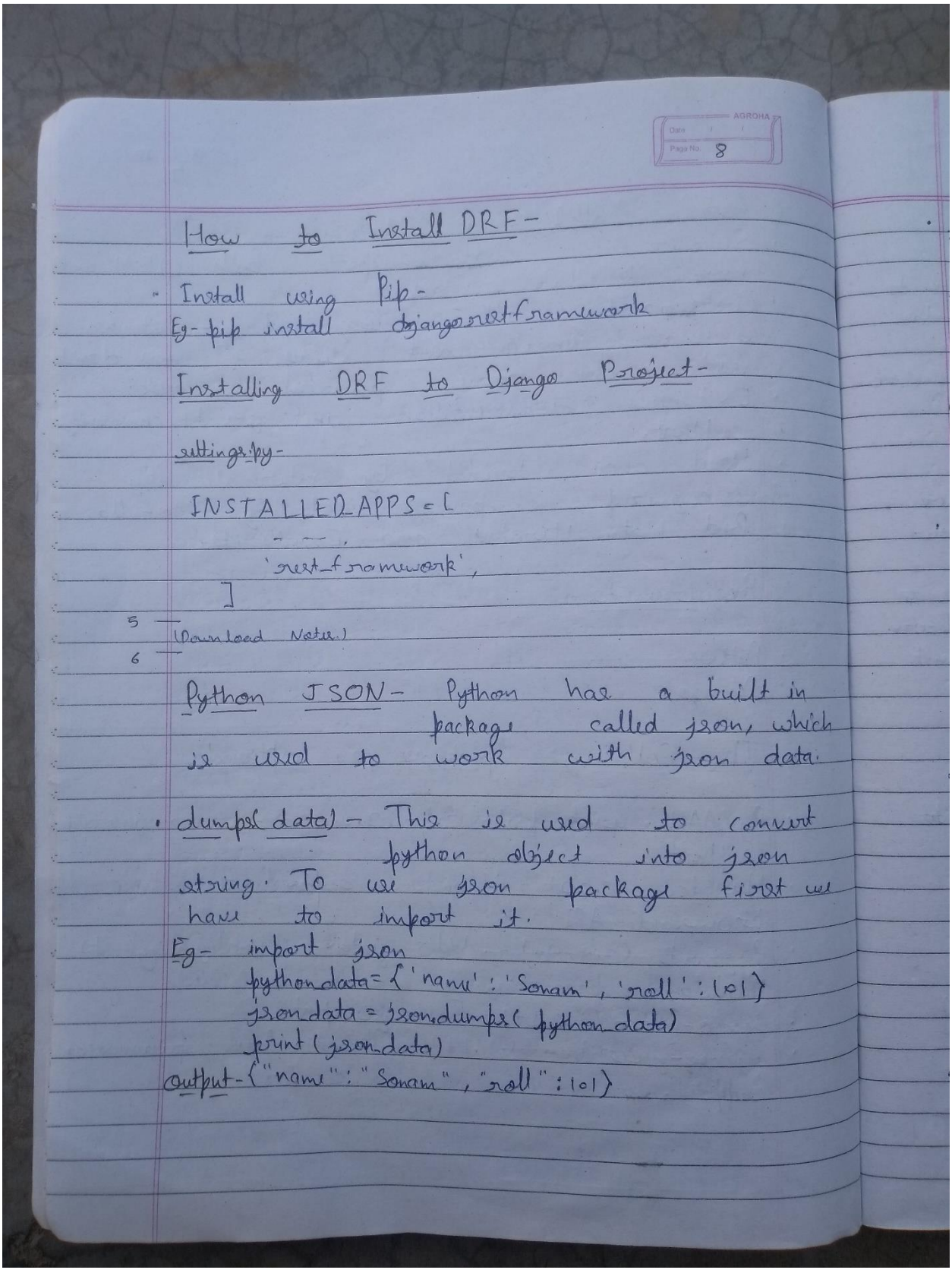
for OAuth1 and OAuth2.
- Serialization that supports both ORM and non-ORM data sources.
- Customizable all the way down - just use regular function-based views if you don't need the more powerful features.
- Extensive documentation, and great community support.
- Used and trusted by internationally recognized companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Requirements-

- Python
- Django

The following packages are optional-

- PyYAML, uritemplate (5.1+, 3.0.0+) - Schema generation support.
- Markdown (3.0.0+) - Markdown support for the browsable API
- Pygments (2.4.0+) - Add syntax highlighting to Markdown processing.
- django-filter (1.0.1+) - Filtering support
- django-guardian (1.1.1+) - Object level permission support.



How to Install DRF-

- Install using Pip-

Eg- pip install djangorestframework

Installing DRF to Django Project-

settings.py-

```
INSTALLED_APPS = [
    ...,
    'rest_framework',
]
```

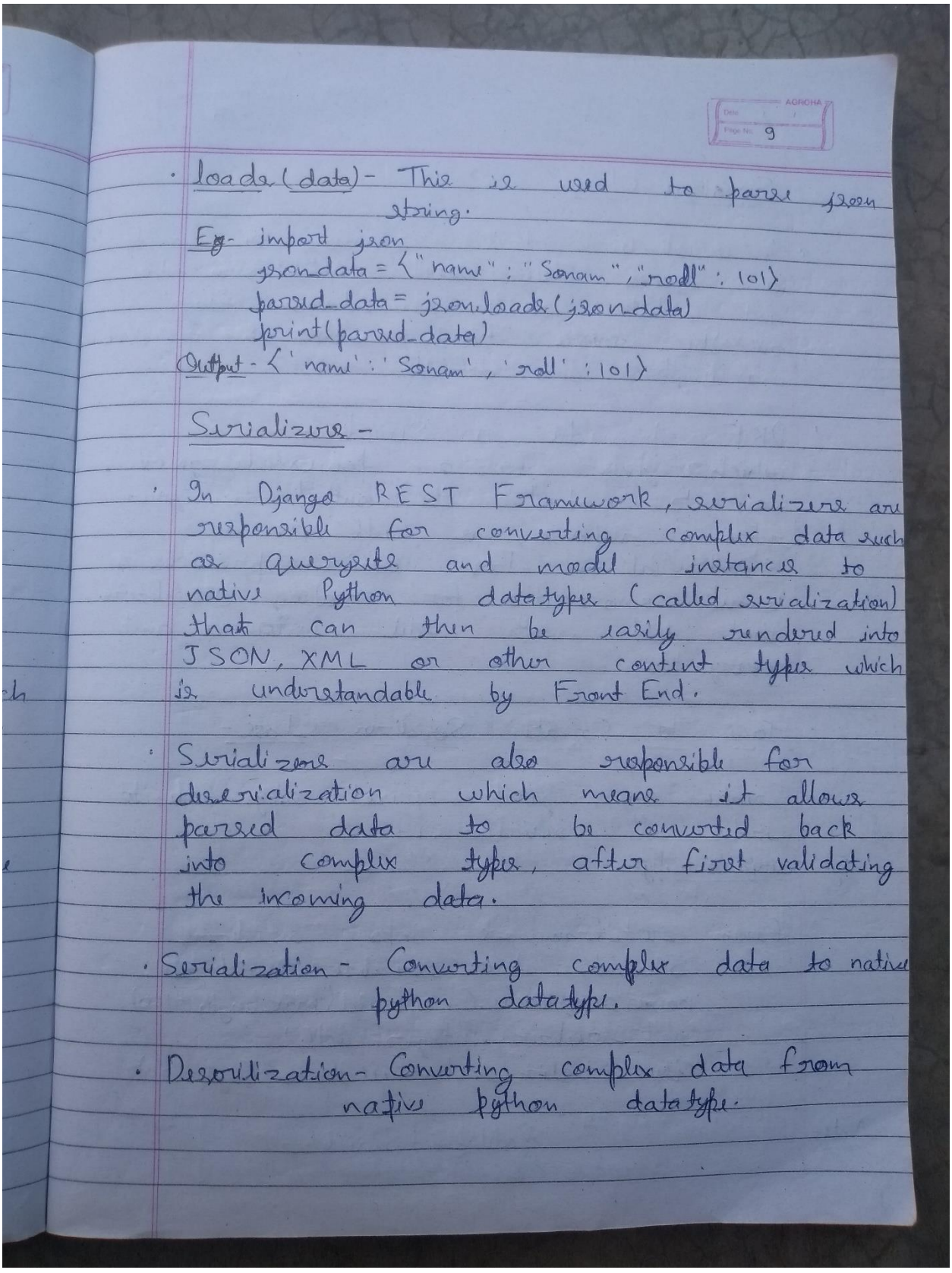
(Download Notes.)

Python JSON- Python has a built in package called json, which is used to work with json data.

- dumps(data) - This is used to convert python object into json string. To use json package first we have to import it.

Eg- import json
python_data = {'name': 'Sonam', 'roll': 101}
json_data = json.dumps(python_data)
print(json_data)

output- {"name": "Sonam", "roll": 101}



- loads(data) - This is used to parse json string.

Eg- import json
json_data = {"name": "Sonam", "roll": 101}
parsed_data = json.loads(json_data)
print(parsed_data)

Output - {'name': 'Sonam', 'roll': 101}

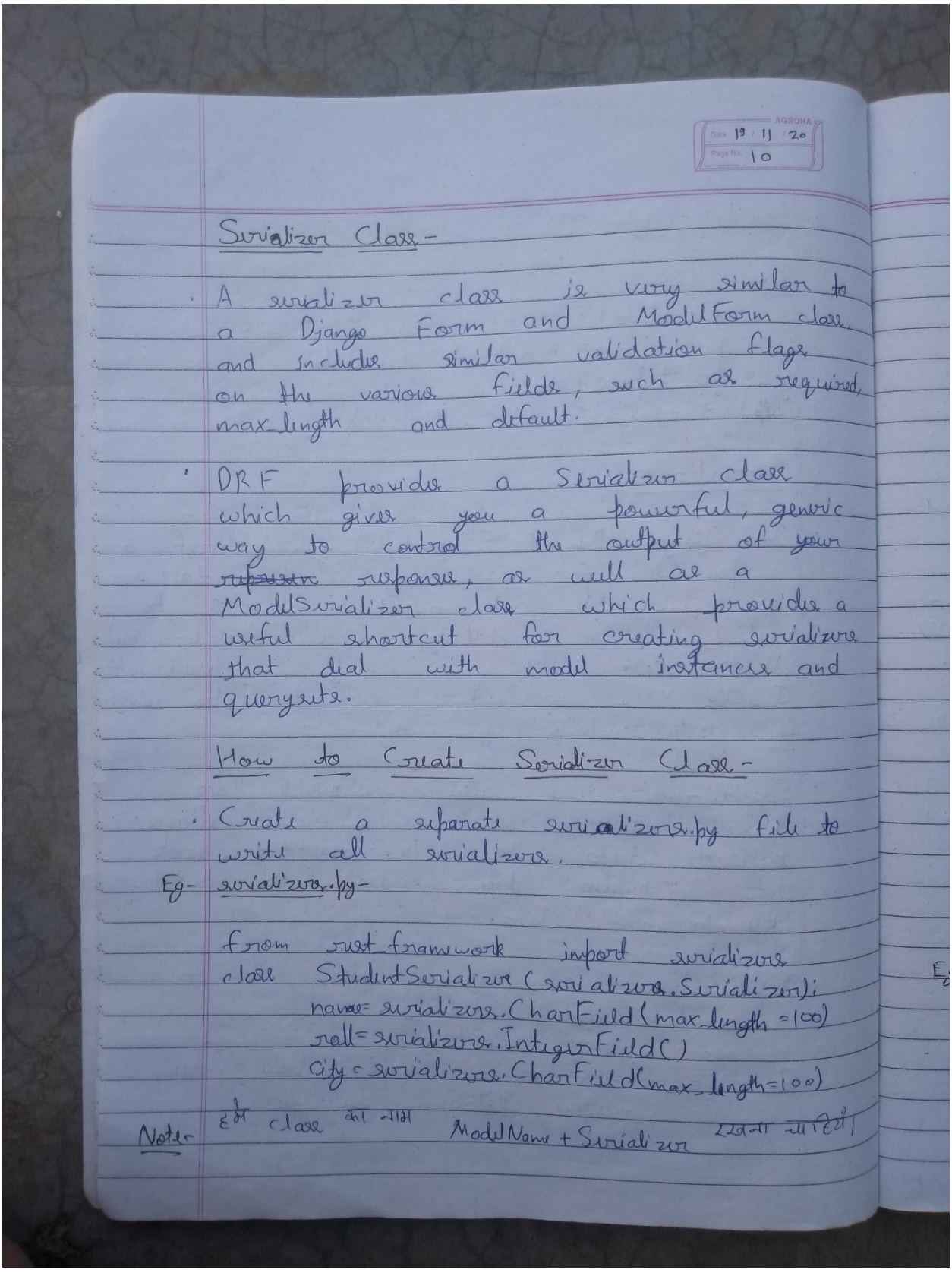
Serializers -

- In Django REST Framework, serializers are responsible for converting complex data such as querysets and model instances to native Python datatypes (called serialization) that can then be easily rendered into JSON, XML or other content types which is understandable by Front End.

- Serializers are also responsible for deserialization which means it allows parsed data to be converted back into complex types, after first validating the incoming data.

- Serialization - Converting complex data to native python datatype.

- Deserialization - Converting complex data from native python datatype.



Date 19/11/20
Page No. 10

Serializer Class -

- A serializer class is very similar to a Django Form and ModelForm class, and includes similar validation flags on the various fields, such as required, max_length and default.

- DRF provide a Serializer class which give you a powerful, generic way to control the output of your ~~reponse~~ response, as well as a ModelSerializer class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

How to Create Serializer Class -

- Create a separate serializers.py file to write all serializers.

Eg- serializers.py -

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name= serializers.CharField(max_length=100)
    roll= serializers.IntegerField()
    city= serializers.CharField(max_length=100)
```

Note- इस class का नाम ModelName + Serializer रखना चाहिए।

· models.py -

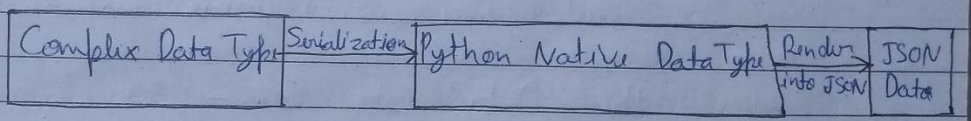
```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length = 100)
    roll = models.IntegerField()
    city = models.CharField(max_length = 100)
```

· Run makemigrations and migrate command.

ID	NAME	ROLL	CITY	
1	Sonam	101	Ranchi	← Model obj 1
2	Rahul	102	Bokaro	← Model obj 2
3	Ram	103	Jaunpur	← Model obj 3

↓↓

JSON Data



Serialization - The process of converting complex data such as queryset and model instances to native python datatypes are called as Serialization in DRF.

Eg- For model Instances

· Create model instance stu -

```
stu = Student.objects.get(id=1)
```

· Converting model instance stu to python Dict/ Serializing object -

```
serializer = StudentSerializer(stu)
```

Eg For QuerySet -

- Creating Query Set -
  stu = Student.objects.all()

- Converting Query Set stu to List of Python Dict / Serializing Query Set -
  serializer = StudentSerializer(stu, many=True)

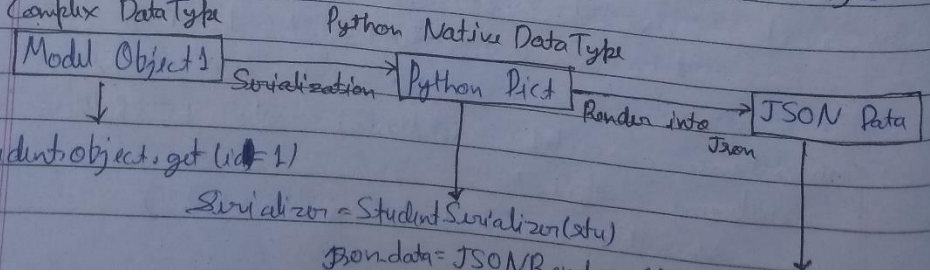
serializer.data - This is the serialized data.
Syntax - serializer.data.

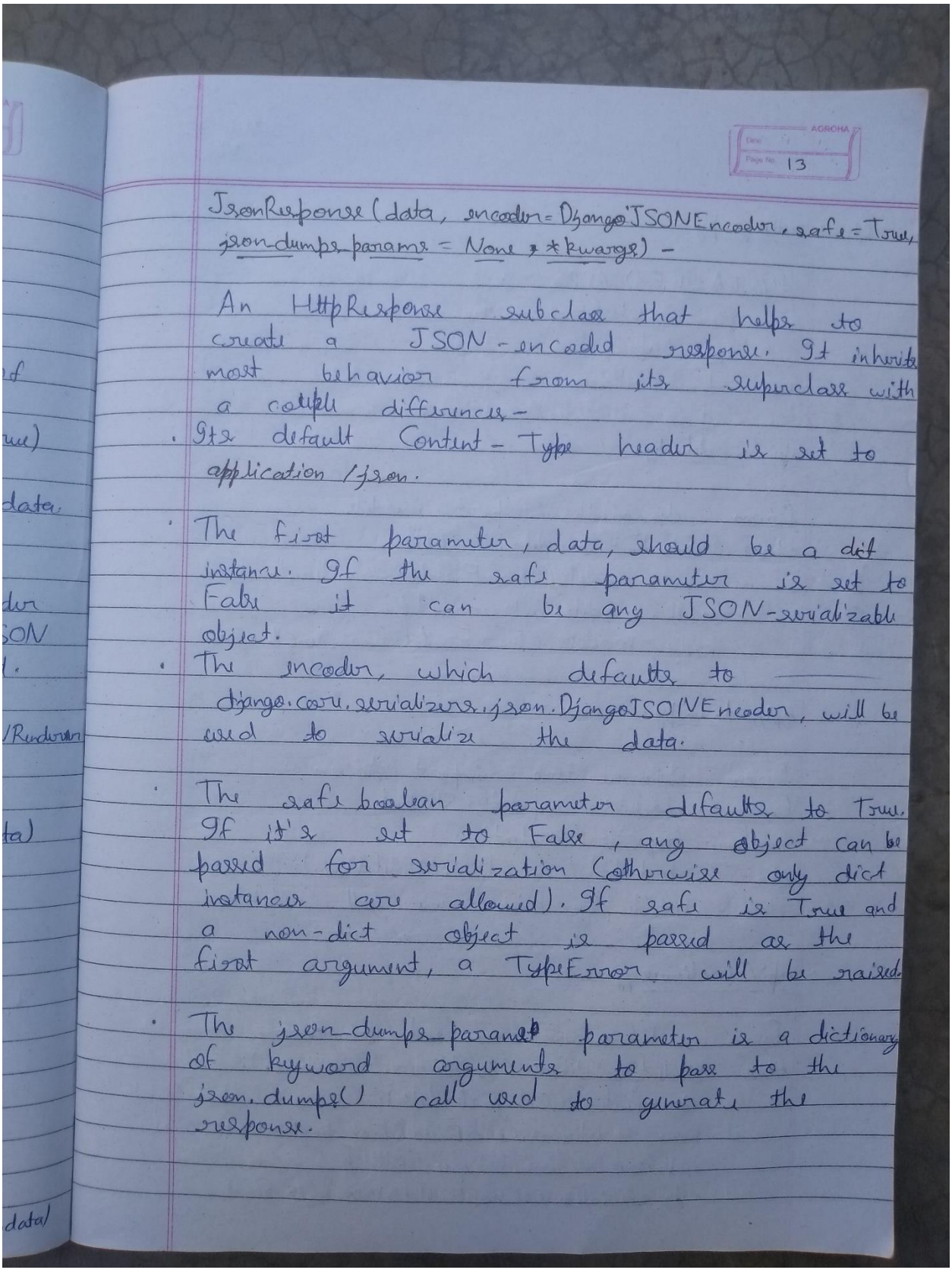
JSONRenderer - This is used to render Serialized data into JSON which is understandable by Front End.

- Import JSONRenderer -
  from rest_framework.renderers import JSONRenderer

- Render the Data into Json -
  json_data = JSONRenderer().render(serializer.data)

ID	NAME	ROLL	CITY	
1	Sonam	101	Ranchi	← Model Object 1
2	Rahul	102	Ranchi	← Model Object 2
3	Raj	103	Bokaro	← Model Object 3

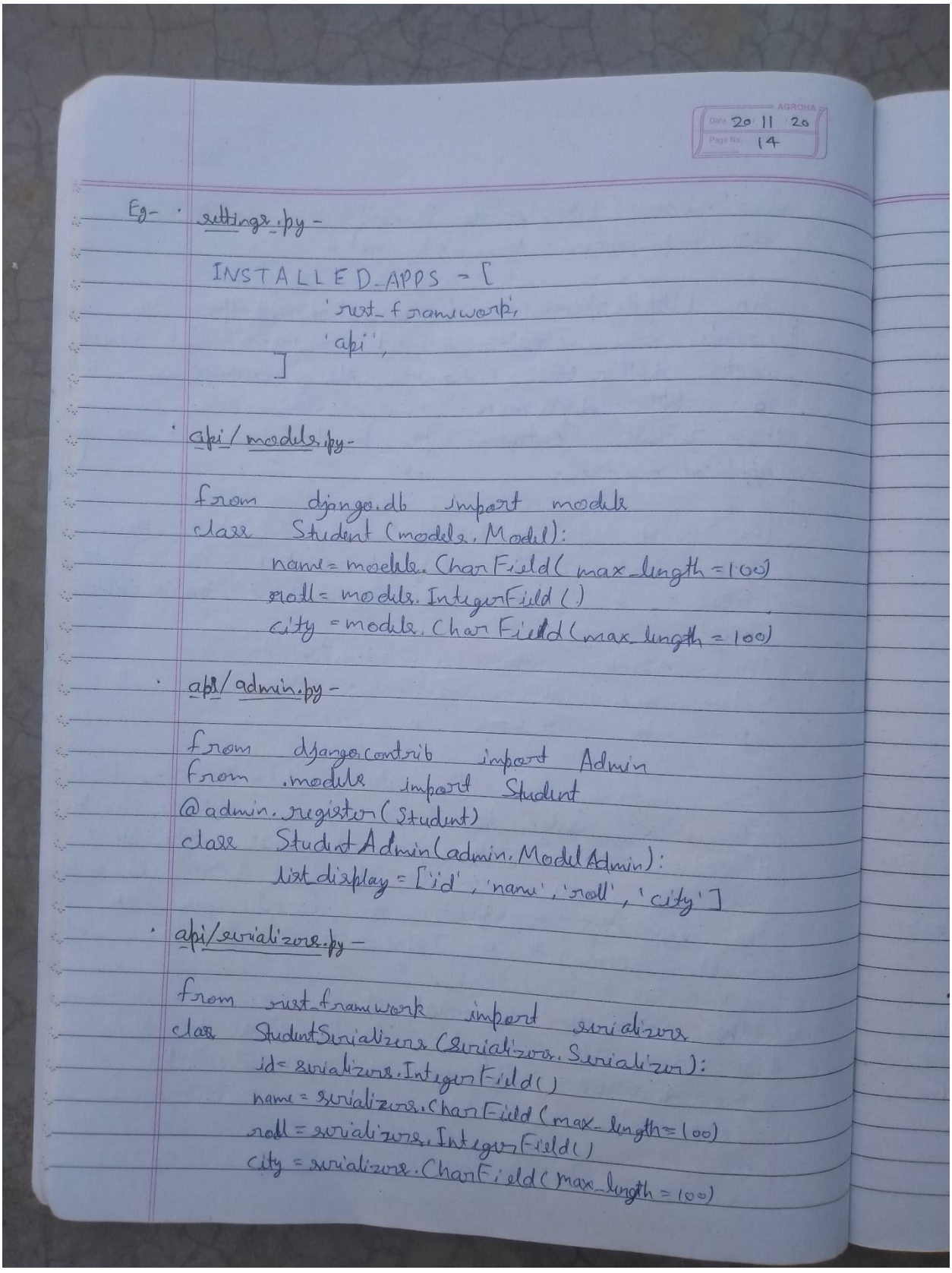




JsonResponse(data, encoder=DjangoJSONEncoder, safe=True, json_dumps_params=None, **kwargs) -

An HttpResponse subclass that helps to create a JSON-encoded response. It inherits most behavior from its superclass with a couple differences -

- Its default Content-Type header is set to application/json.
- The first parameter, data, should be a dict instance. If the safe parameter is set to False it can be any JSON-serializable object.
- The encoder, which defaults to django.core.serializers.json.DjangoJSONEncoder, will be used to serialize the data.
- The safe boolean parameter defaults to True. If it's set to False, any object can be passed for serialization (otherwise only dict instances are allowed). If safe is True and a non-dict object is passed as the first argument, a TypeError will be raised.
- The json_dumps_params parameter is a dictionary of keyword arguments to pass to the json.dumps() call used to generate the response.



Eg- · settings.py -

```
INSTALLED_APPS = [
        'rest_framework',
        'api',
    ]
```

· api/models.py -

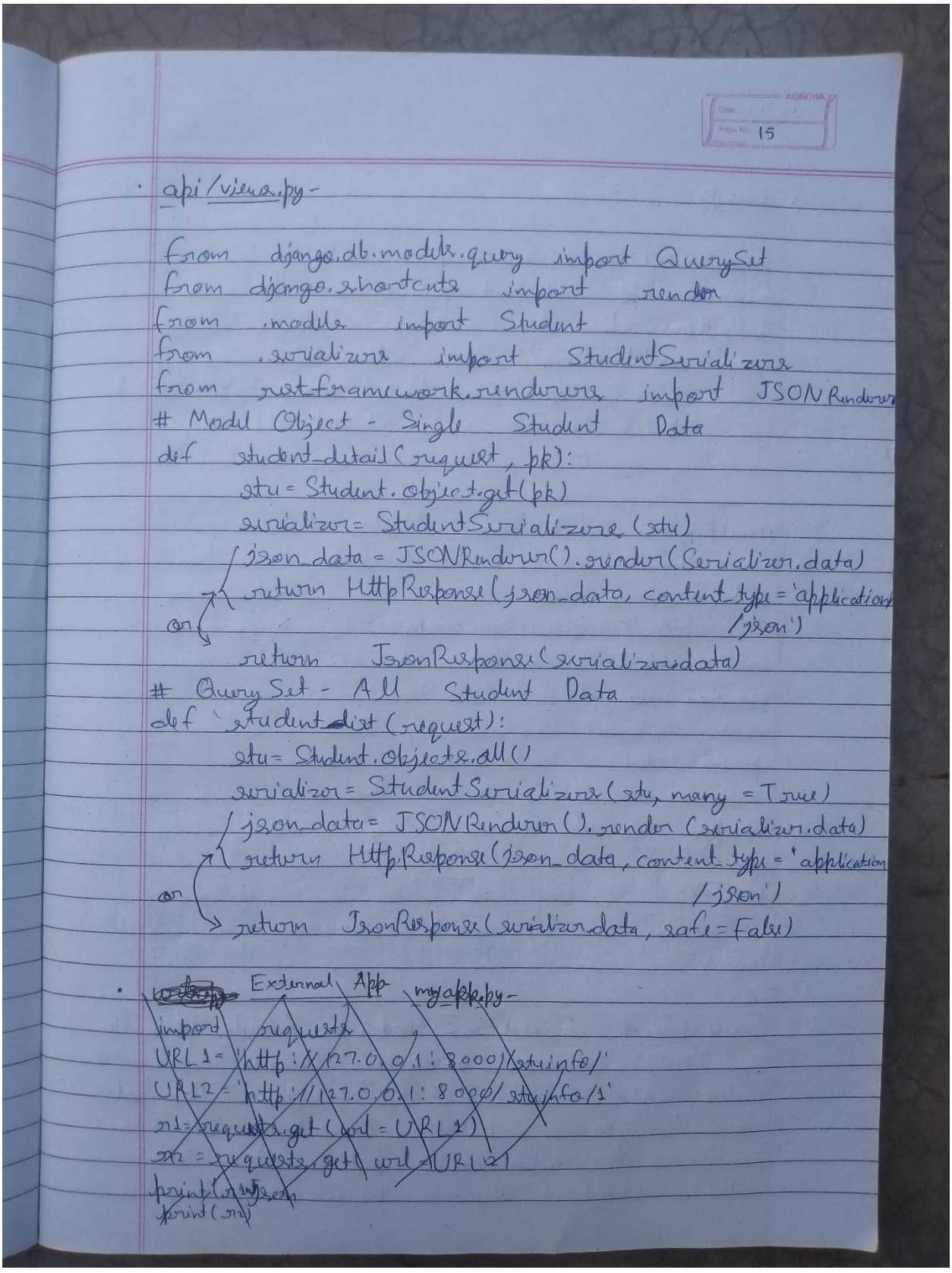
```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=100)
    roll = models.IntegerField()
    city = models.CharField(max_length=100)
```

· api/admin.py -

```
from django.contrib import Admin
from .models import Student
@admin.register(Student)
class StudentAdmin(admin.ModelAdmin):
    list_display = ['id', 'name', 'roll', 'city']
```

· api/serializers.py -

```
from rest_framework import serializers
class StudentSerializers(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

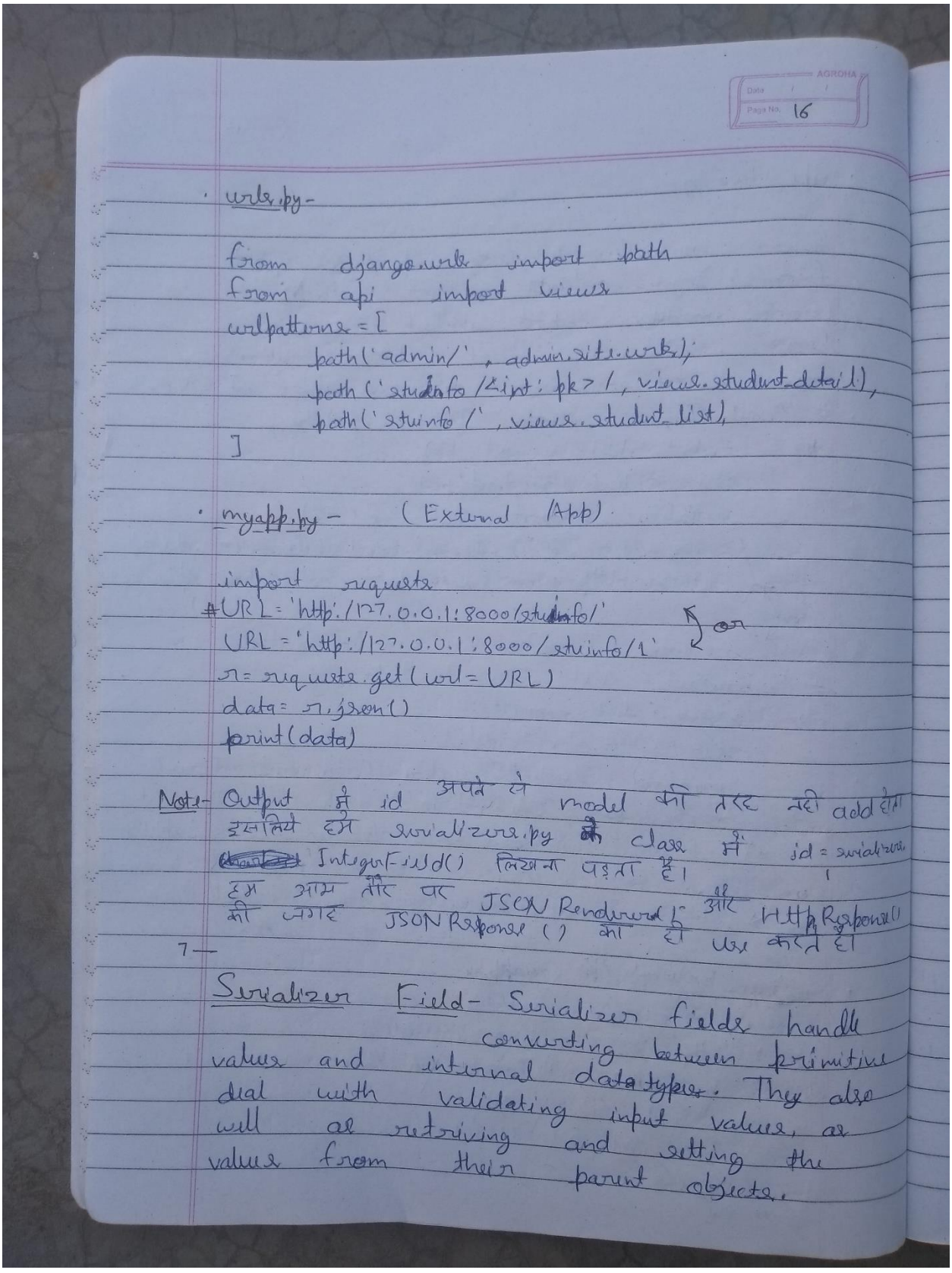


- api/views.py -

```
from django.db.models.query import QuerySet
from django.shortcuts import render
from .models import Student
from .serializers import StudentSerializers
from rest_framework.renderers import JSONRenderer
# Model Object - Single Student Data
def student_detail(request, pk):
    stu = Student.objects.get(pk)
    serializer = StudentSerializers(stu)
    json_data = JSONRenderer().render(serializer.data)
    return HttpResponse(json_data, content_type = 'application/json')
or
    return JsonResponse(serializer.data)
# QuerySet - All Student Data
def student_list(request):
    stu = Student.objects.all()
    serializer = StudentSerializers(stu, many = True)
    json_data = JSONRenderer().render(serializer.data)
    return HttpResponse(json_data, content_type = 'application/json')
or
    return JsonResponse(serializer.data, safe = False)
```

- External App myapp.py -

```
import requests
URL1 = 'http://127.0.0.1:8000/stuinfo/'
URL2 = 'http://127.0.0.1:8000/stuinfo/1'
r1 = requests.get(url = URL1)
r2 = requests.get(url = URL2)
print(r1.json())
print(r2)
```



Data / / Page No. 16

- urls.py -

```
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('stuinfo/<int:pk>/', views.student_detail),
    path('stuinfo/', views.student_list),
]
```

- myapp.py - (External /App)

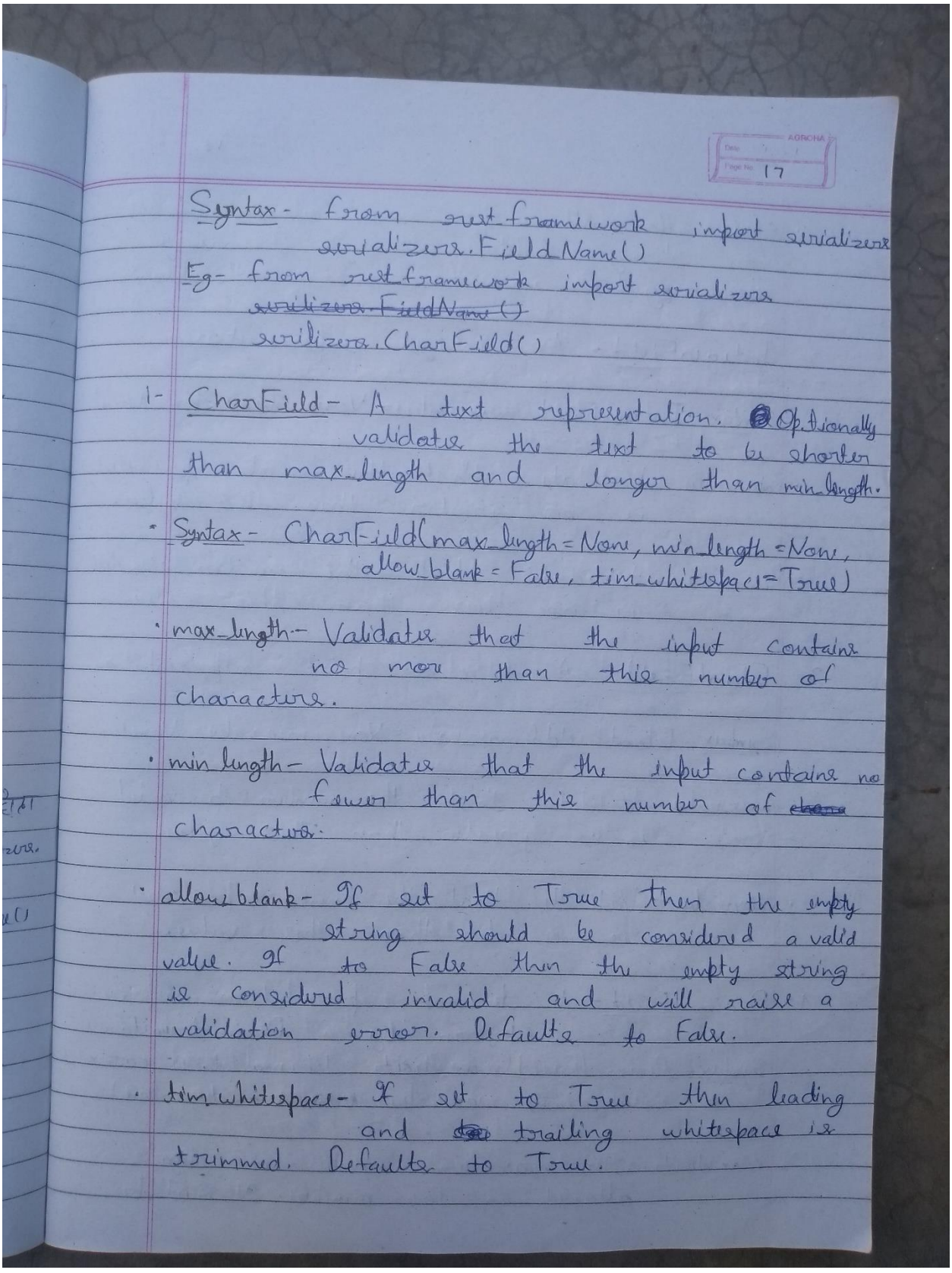
```
import requests
#URL = 'http://127.0.0.1:8000/stuinfo/'
URL = 'http://127.0.0.1:8000/stuinfo/1'
r = requests.get(url=URL)
data = r.json()
print(data)
```

Note- Output में id अपने से model की तरह नहीं add होता इसलिये हमें serializers.py की class में id = serializers.IntegerField() लिखना पड़ता है।

हम आम तौर पर JSONRenderer और HttpResponse() की जगह JSONResponse() का ही use करते है।

7—

Serializer Field- Serializer fields handle converting between primitive values and internal datatypes. They also deal with validating input values, as well as retriving and setting the values from their parent objects.



Syntax - from rest_framework import serializers
serializers.FieldName()
Eg - from rest_framework import serializers
~~serilizers.FieldName()~~
serilizers.CharField()

1- CharField - A text representation. Optionally validates the text to be shorter than max_length and longer than min_length.

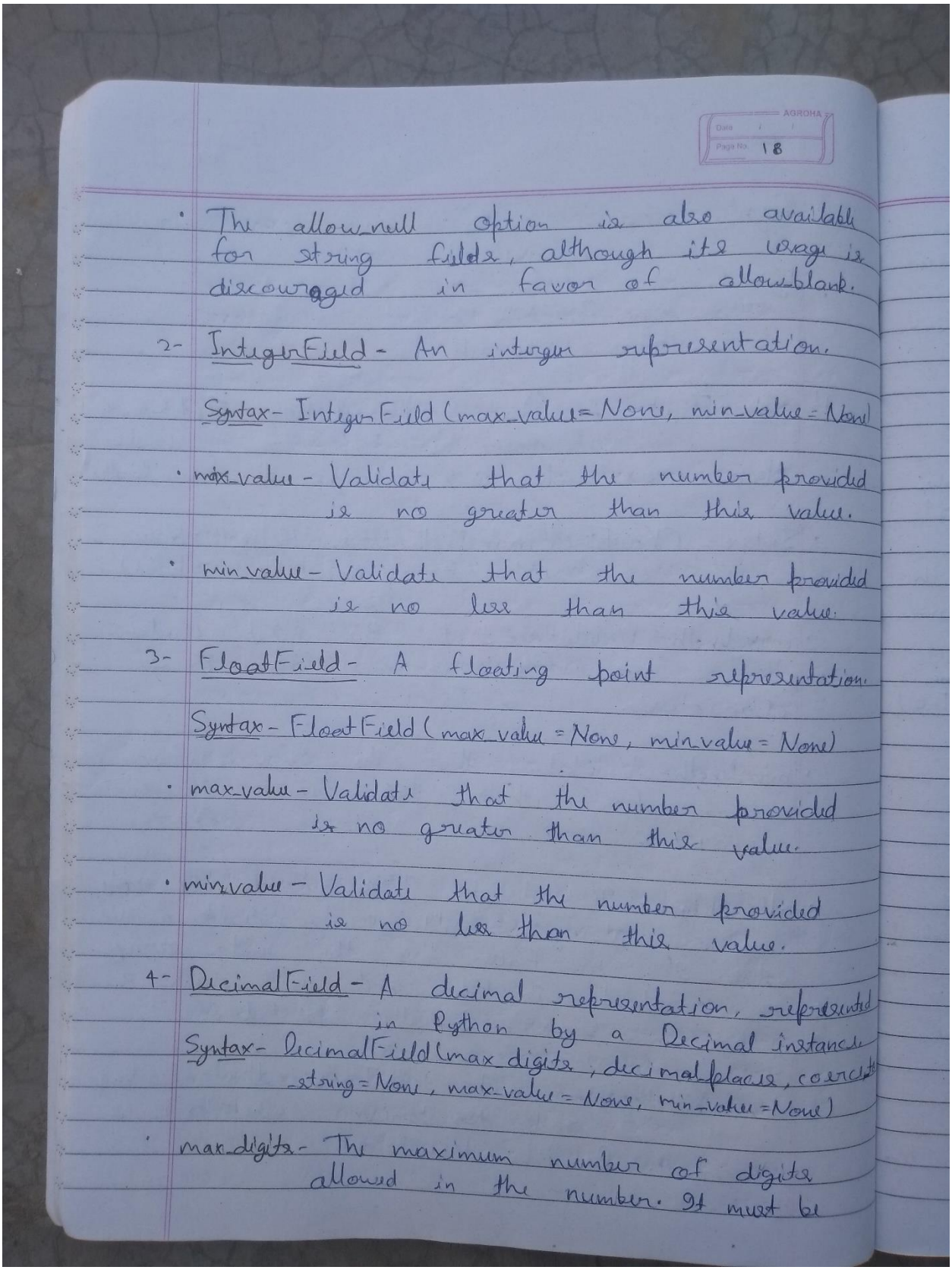
- Syntax - CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)

- max_length - Validates that the input contains no more than this number of characters.

- min_length - Validates that the input contains no fewer than this number of characters.

- allow_blank - If set to True then the empty string should be considered a valid value. If set to False then the empty string is considered invalid and will raise a validation error. Defaults to False.

- trim_whitespace - If set to True then leading and trailing whitespace is trimmed. Defaults to True.



- The allow_null option is also available for string fields, although its usage is discouraged in favor of allow_blank.

2- IntegerField - An integer representation.

Syntax - IntegerField(max_value=None, min_value=None)

- max_value - Validates that the number provided is no greater than this value.
- min_value - Validates that the number provided is no less than this value.

3- FloatField - A floating point representation.

Syntax - FloatField(max_value=None, min_value=None)

- max_value - Validates that the number provided is no greater than this value.
- min_value - Validates that the number provided is no less than this value.

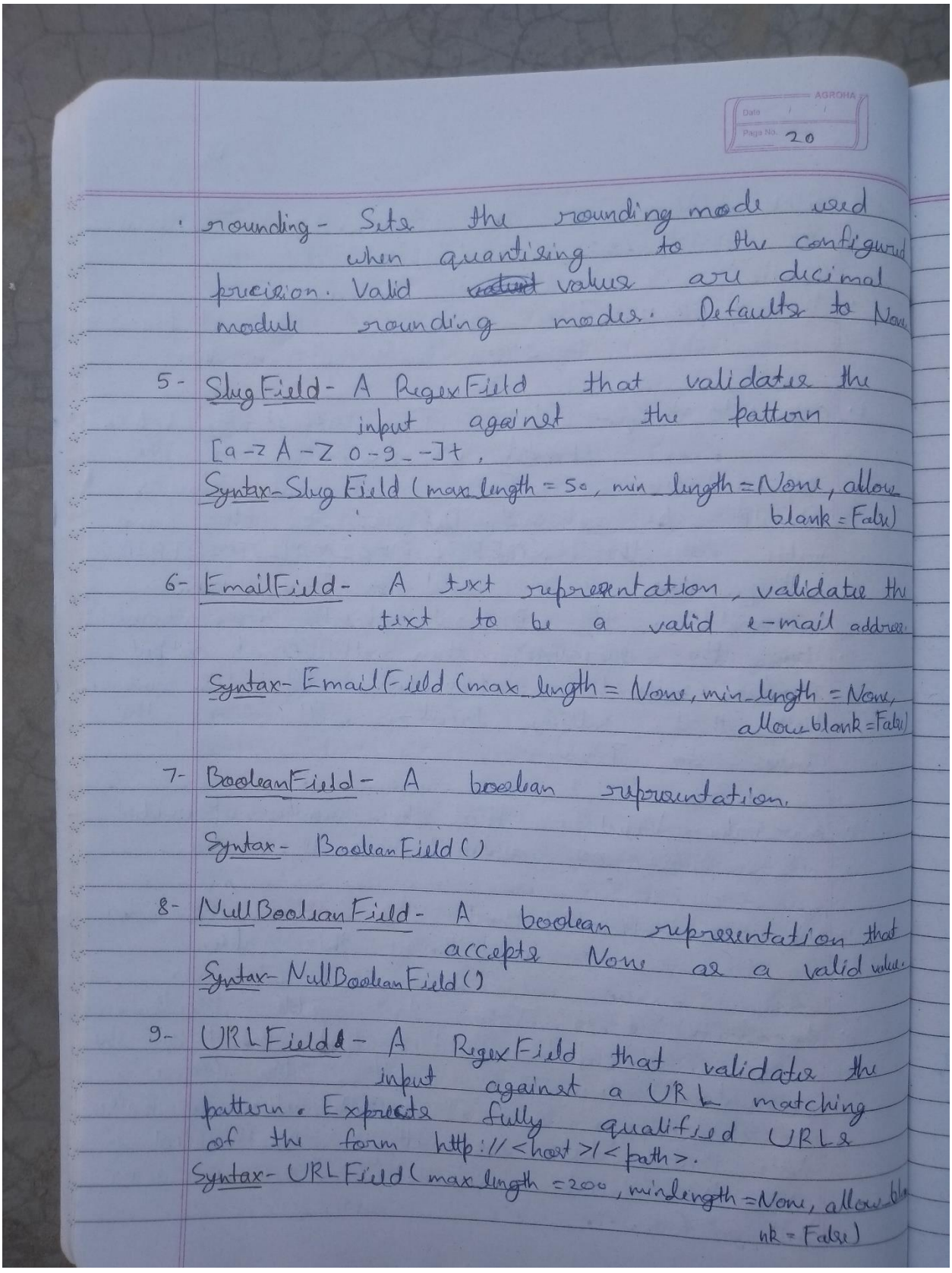
4- DecimalField - A decimal representation, represented in Python by a Decimal instance.

Syntax - DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None)

- max_digits - The maximum number of digits allowed in the number. It must be

either None or an integer grater than or equal to decimal_places.

- decimal_places - The number of decimal places to store with the number.
- coerce_to_string - Set to True if string values should be returned for the representation, or False if Decimal objects should be returned. Defaults to the same value as the COERCE_DECIMAL_TO_STRING settings key, which will be True unless overridden. If Decimal objects are returned by the serializer, then the final output format will be determined by the renderer. Note that setting localize will force the value to True.
- max_value - Validate that the number provided is no greater than this value.
- min_value - Validate that the number provided is no less than this value.
- localize - Set to True to enable localization of input and output based on the current locale. This will also force coerce_to_string to True. Defaults to False. Note that data formatting is enabled if you have set USE_L10N = True in your settings file.



- rounding - Sets the rounding mode used when quantising to the configured precision. Valid ~~values~~ values are decimal module rounding modes. Defaults to None.

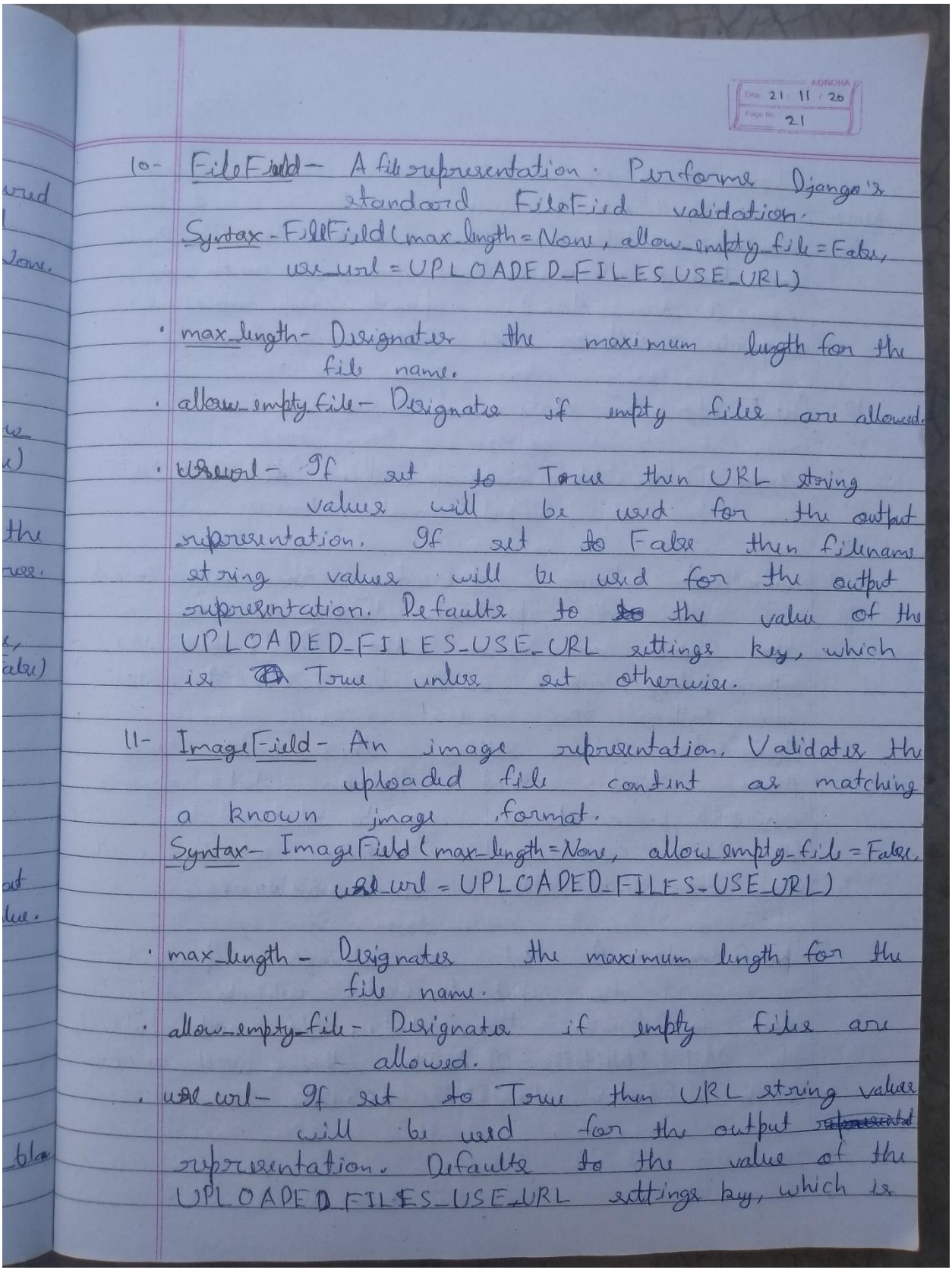
5- SlugField - A RegexField that validates the input against the pattern [a-z A-Z 0-9 _ -]+.
Syntax - SlugField(max_length=50, min_length=None, allow_blank=False)

6- EmailField - A text representation, validates the text to be a valid e-mail address.
Syntax - EmailField(max_length=None, min_length=None, allow_blank=False)

7- BooleanField - A boolean representation.
Syntax - BooleanField()

8- NullBooleanField - A boolean representation that accepts None as a valid value.
Syntax - NullBooleanField()

9- URLField - A RegexField that validates the input against a URL matching pattern. Expects fully qualified URLs of the form http://<host>/<path>.
Syntax - URLField(max_length=200, min_length=None, allow_blank=False)



10- FileField - A file representation. Performs Django's standard FileField validation.

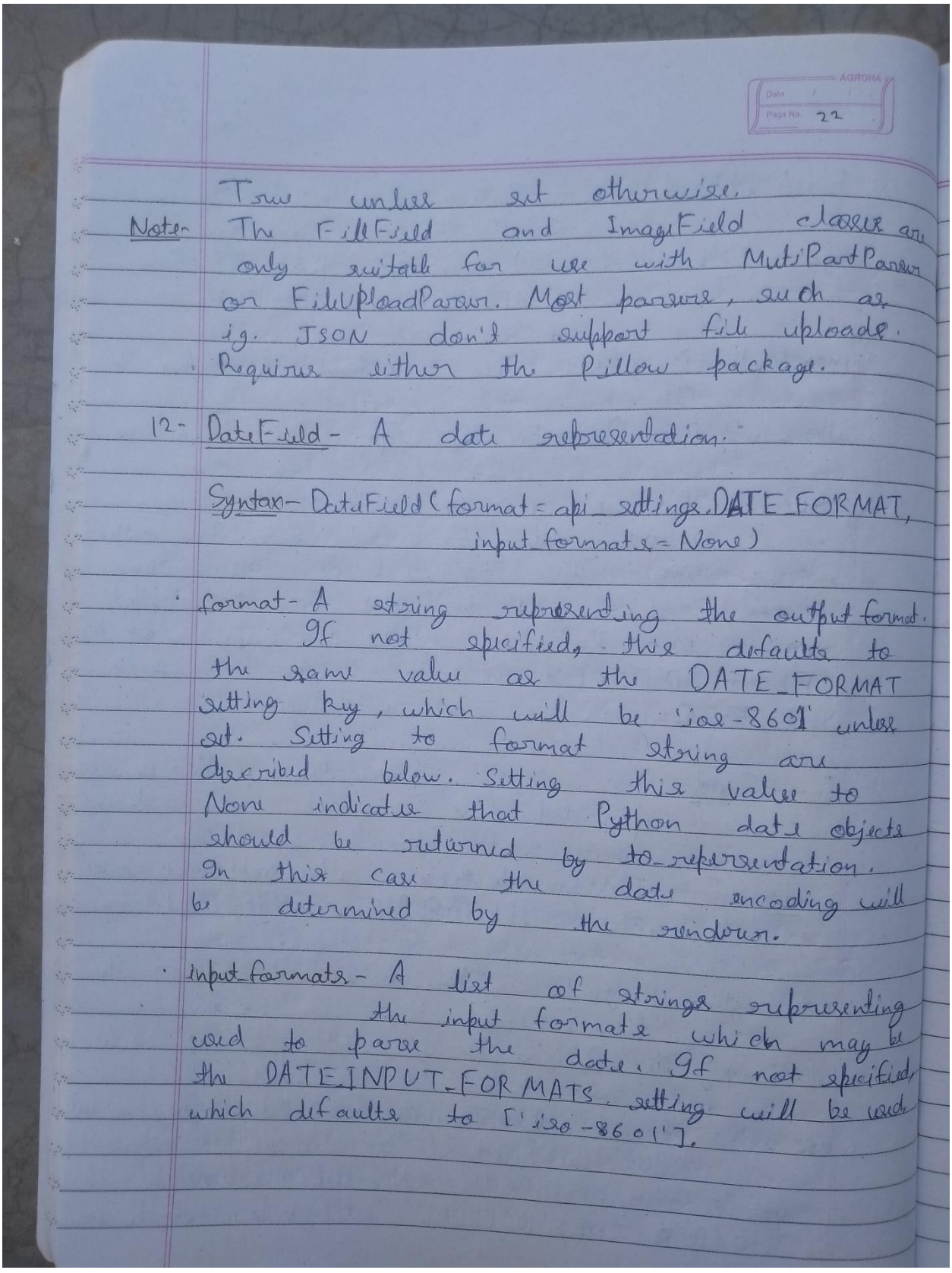
Syntax - FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)

- max_length - Designates the maximum length for the file name.
- allow_empty_file - Designates if empty files are allowed.
- use_url - If set to True then URL string values will be used for the output representation. If set to False then filename string values will be used for the output representation. Defaults to the value of the UPLOADED_FILES_USE_URL settings key, which is True unless set otherwise.

11- ImageField - An image representation. Validates the uploaded file content as matching a known image format.

Syntax - ImageField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)

- max_length - Designates the maximum length for the file name.
- allow_empty_file - Designates if empty files are allowed.
- use_url - If set to True then URL string values will be used for the output representation. Defaults to the value of the UPLOADED_FILES_USE_URL settings key, which is



True unless set otherwise.

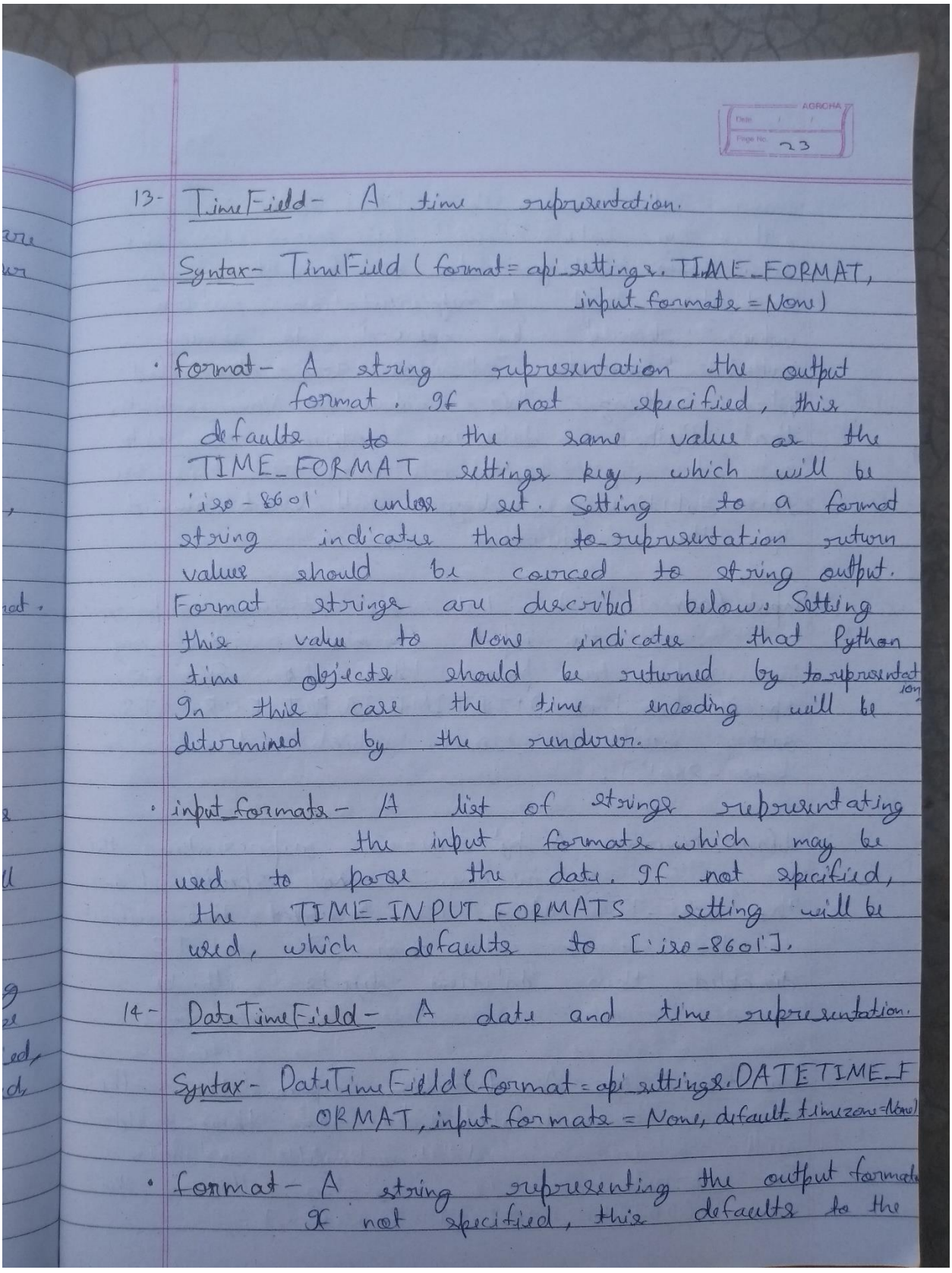
Note- The FileField and ImageField classes are only suitable for use with MultiPartParser or FileUploadParser. Most parsers, such as e.g. JSON don't support file uploads. Requires either the Pillow package.

12- DateField - A date representation.

Syntax- DateField(format=api_settings.DATE_FORMAT, input_formats=None)

- format - A string representing the output format. If not specified, this defaults to the same value as the DATE_FORMAT setting key, which will be 'iso-8601' unless set. Setting to format string are described below. Setting this value to None indicates that Python date objects should be returned by to_representation. In this case the date encoding will be determined by the renderer.

- input_formats - A list of strings representing the input formats which may be used to parse the date. If not specified, the DATE_INPUT_FORMATS setting will be used which defaults to ['iso-8601'].



13- TimeField - A time representation.

Syntax- TimeField (format = api_settings.TIME_FORMAT, input_formats = None)

- format - A string representation the output format. If not specified, this defaults to the same value as the TIME_FORMAT settings key, which will be 'iso-8601' unless set. Setting to a format string indicates that to_representation return values should be coerced to string output. Format strings are described below. Setting this value to None indicates that Python time objects should be returned by to_representation. In this case the time encoding will be determined by the renderer.

- input_formats - A list of strings representating the input formats which may be used to parse the date. If not specified, the TIME_INPUT_FORMATS setting will be used, which defaults to ['iso-8601'].

14- DateTimeField - A date and time representation.

Syntax- DateTimeField (format = api_settings.DATETIME_FORMAT, input_formats = None, default_timezone = None)

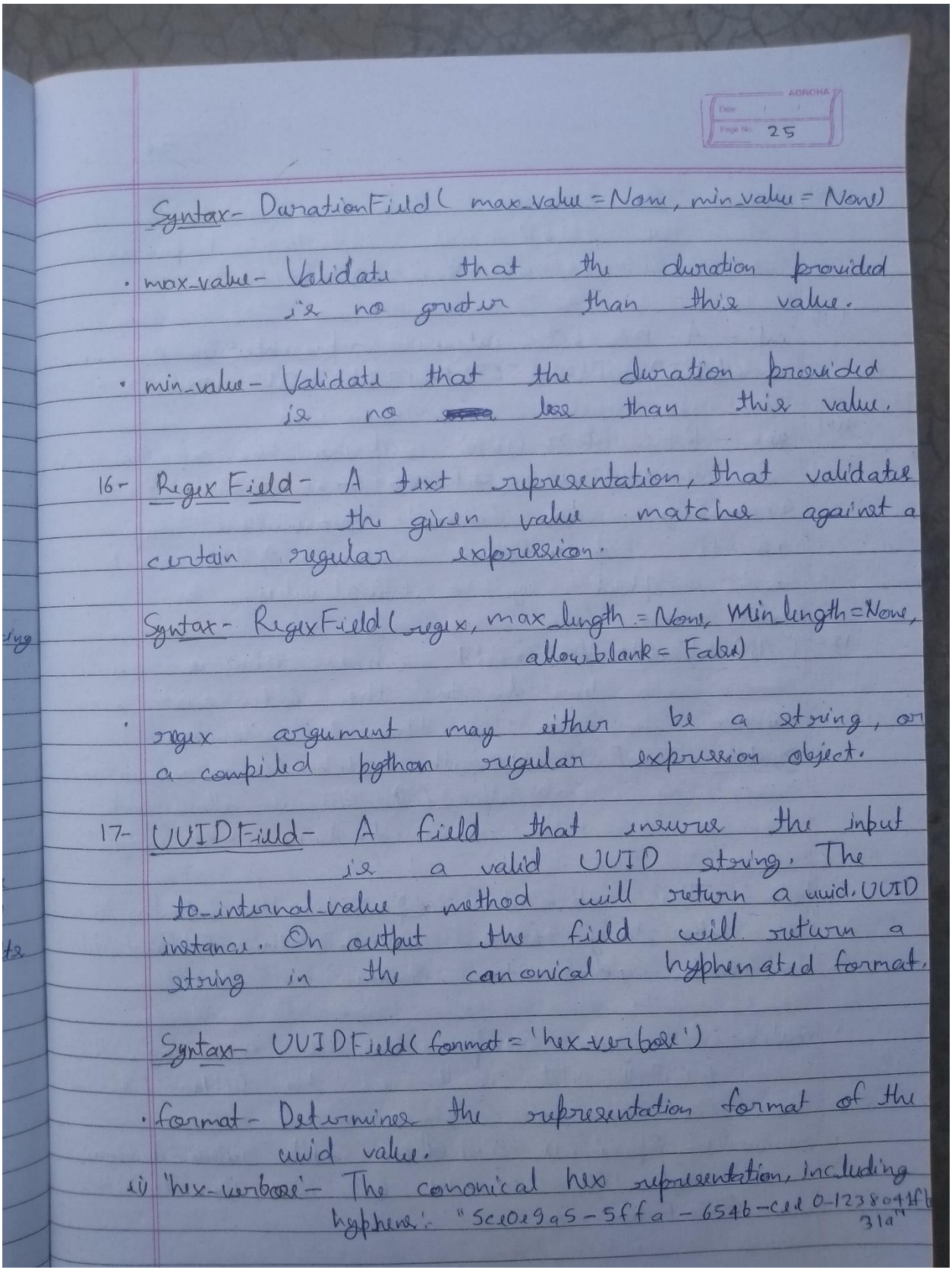
- format - A string representing the output format. If not specified, this defaults to the

same value as the DATETIME_FORMAT settings key, which will be 'iso-8601' unless set. ~~String~~ Setting to a format string indicates that to_representation return values should be coerced to string output. Format strings are described below. Setting this value to None indicates that Python datetime objects should be returned by to_representation. In this case the datetime encoding will be determined by the renderer.

- input_formats - A list of strings representing the input formats which may be used to parse the date. If not specified, the DATETIME_INPUT_FORMATS setting will be used, which defaults to ['iso-8601'].

- default_timezone - A pytz.timezone representing the timezone. If not specified and the USE_TZ setting is enabled, this defaults to the current timezone. If USE_TZ is disabled, then datetime objects will be naive.

15- Duration Field - A duration representation. The validated_data for these fields will contain a datetime.timedelta instance. The representation is a string following this format '[DD] [HH:[MM:]]ss[.uuuuuu]'.



Syntax- DurationField(max_value = None, min_value = None)

- max_value - Validate that the duration provided is no greater than this value.
- min_value - Validate that the duration provided is no less than this value.

16- Regex Field - A text representation, that validates the given value matches against a certain regular expression.

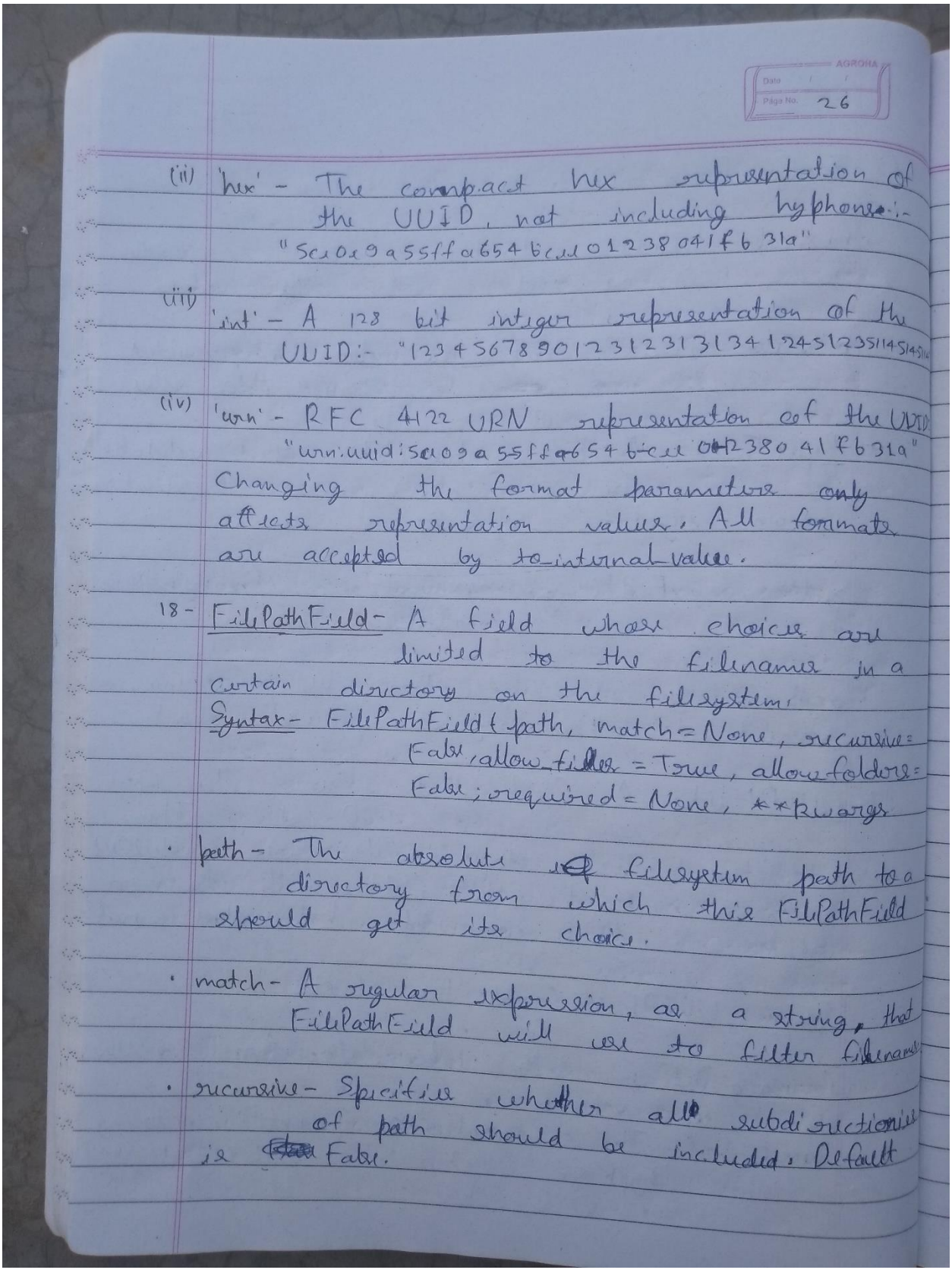
Syntax - RegexField(regex, max_length = None, min_length = None, allow_blank = False)

- regex argument may either be a string, or a compiled python regular expression object.

17- UUIDField - A field that ensures the input is a valid UUID string. The to_internal_value method will return a uuid.UUID instance. On output the field will return a string in the canonical hyphenated format.

Syntax - UUIDField(format = 'hex_verbose')

- format - Determines the representation format of the uuid value.
  - i) 'hex_verbose' - The canonical hex representation, including hyphens: "5ce0e9a5-5ffa-654b-cee0-1238041fb31a"



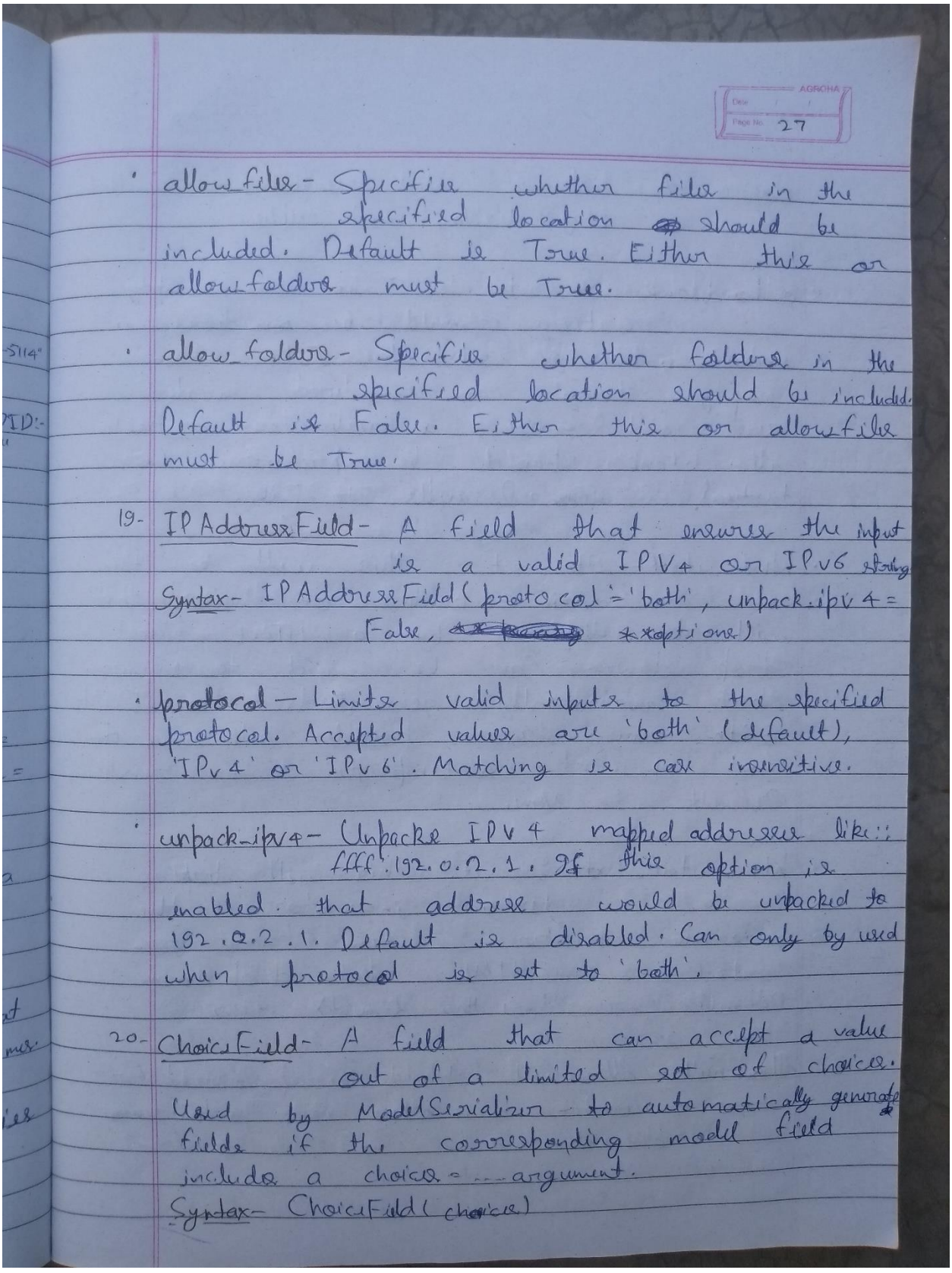
(ii) 'hex' - The compact hex representation of the UUID, not including hyphens:- "5c0a9a55ffa654bcd01238041fb31a"

(iii) 'int' - A 128 bit integer representation of the UUID:- "12345678901231231341245123511451..."

(iv) 'urn' - RFC 4122 URN representation of the UUID "urn:uuid:5c0a9a55ffa654bcd012380 41fb31a" Changing the format parameters only affects representation value. All formats are accepted by to_internal_value.

18- FilePathField - A field whose choices are limited to the filenames in a certain directory on the filesystem.
Syntax- FilePathField(path, match=None, recursive=False, allow_files=True, allow_folders=False, required=None, **kwargs)

- path - The absolute filesystem path to a directory from which this FilePathField should get its choices.
- match - A regular expression, as a string, that FilePathField will use to filter filenames.
- recursive - Specifies whether all subdirectories of path should be included. Default is False.



- allow_file - Specifies whether files in the specified location should be included. Default is True. Either this or allow_folder must be True.

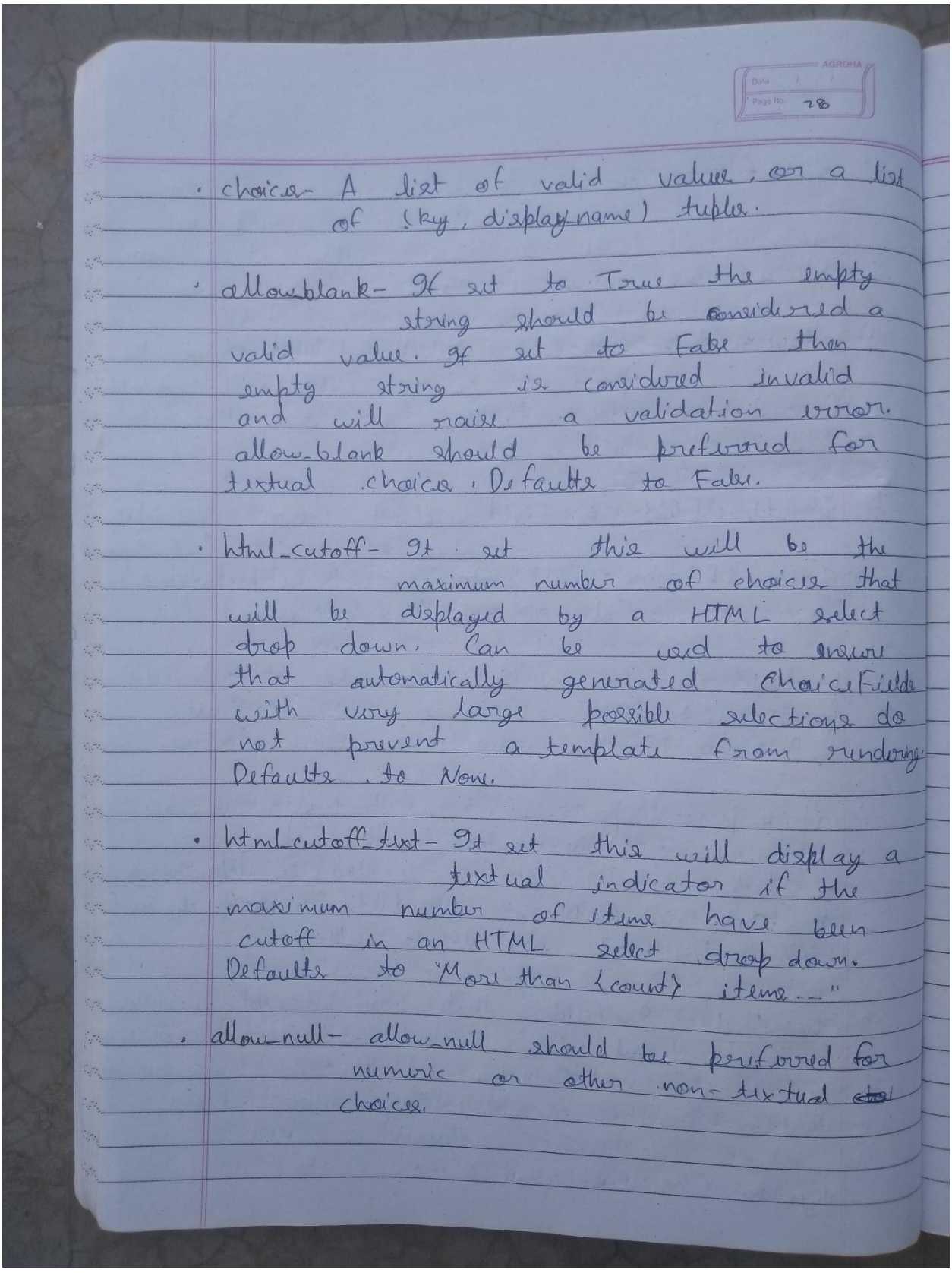
- allow_folder - Specifies whether folders in the specified location should be included. Default is False. Either this or allow_file must be True.

19- IPAddressField - A field that ensures the input is a valid IPV4 or IPv6 string.
Syntax - IPAddressField(protocol='both', unpack_ipv4=False, **options)

- protocol - Limits valid inputs to the specified protocol. Accepted values are 'both' (default), 'IPv4' or 'IPv6'. Matching is case insensitive.

- unpack_ipv4 - Unpacks IPV4 mapped addresses like ::ffff:192.0.2.1. If this option is enabled that address would be unpacked to 192.0.2.1. Default is disabled. Can only by used when protocol is set to 'both'.

20- ChoiceField - A field that can accept a value out of a limited set of choices. Used by ModelSerializer to automatically generate fields if the corresponding model field includes a choices=... argument.
Syntax - ChoiceField(choices)



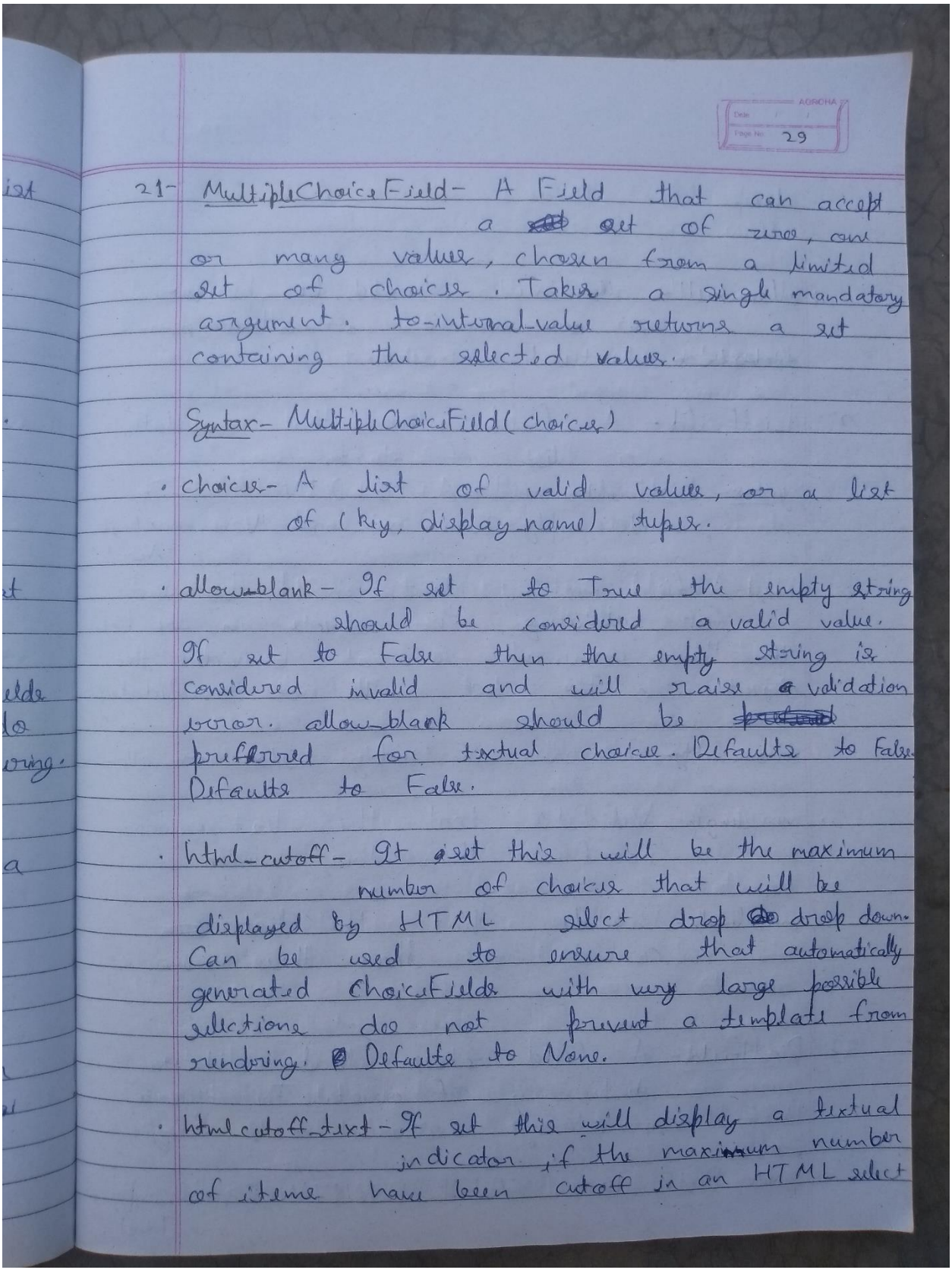
- choices- A list of valid values, or a list of (key, display_name) tuples.

- allow_blank- If set to True the empty string should be considered a valid value. If set to False then empty string is considered invalid and will raise a validation error. allow_blank should be preferred for textual choices. Defaults to False.

- html_cutoff- If set this will be the maximum number of choices that will be displayed by a HTML select drop down. Can be used to ensure that automatically generated ChoiceFields with very large possible selections do not prevent a template from rendering. Defaults to None.

- html_cutoff_text- If set this will display a textual indicator if the maximum number of items have been cutoff in an HTML select drop down. Defaults to "More than {count} items..."

- allow_null- allow_null should be preferred for numeric or other non-textual choices.



21- MultipleChoiceField - A Field that can accept a set of zero, one or many values, chosen from a limited set of choices. Takes a single mandatory argument. to_internal_value returns a set containing the selected values.

Syntax - MultipleChoiceField(choices)

- choices - A list of valid values, or a list of (key, display_name) tuples.
- allow_blank - If set to True the empty string should be considered a valid value. If set to False then the empty string is considered invalid and will raise a validation error. allow_blank should be preferred for textual choices. Defaults to False. Defaults to False.
- html_cutoff - If set this will be the maximum number of choices that will be displayed by HTML select drop down. Can be used to ensure that automatically generated ChoiceFields with very large possible selections do not prevent a template from rendering. Defaults to None.
- html_cutoff_text - If set this will display a textual indicator if the maximum number of items have been cutoff in an HTML select

drop down. Defaults to "More than {count} items ..."

- allow_null - allow_null should be performed for numeric or other non-textual choice.

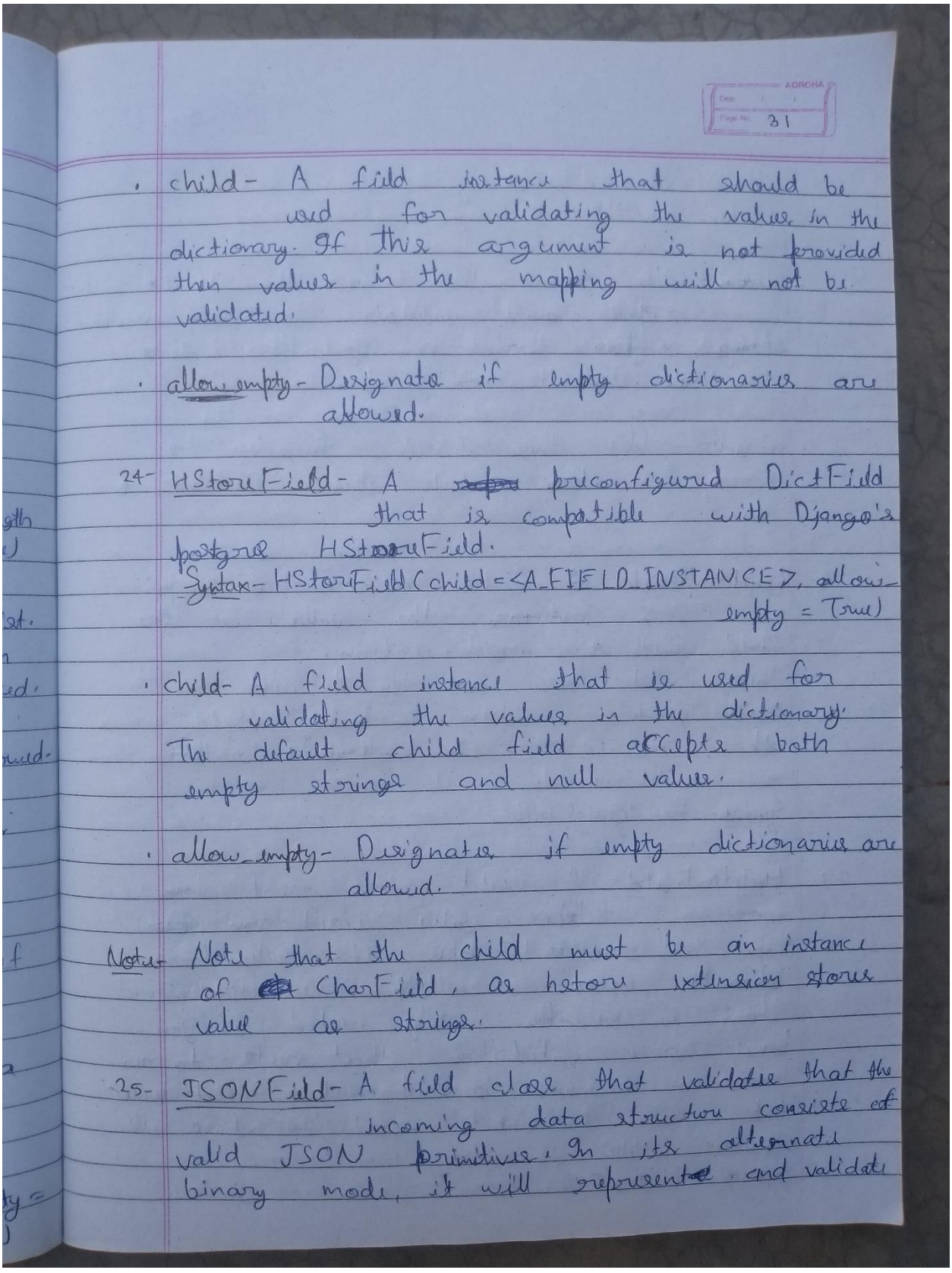
22- ListField - A field class that validates a list of objects.

Syntax - ListField(child=<A_FIELD_INSTANCE>, allow_empty=True, min_length=None, max_length=None)

- child - A field instance that should be used for validating the objects in the list. If this argument is not provided then objects in the list will not be validated.
- allow_empty - Designates if empty lists are allowed.
- min_length - Validates that the list contains no fewer than this number of elements.
- max_length - Validates that the list contains more than this number of elements.

23- DictField - A field class that validates a dictionary of objects. The keys in DictField are always assumed to be string values.

Syntax - DictField(child=<A_FIELD_INSTANCE>, allow_empty=True)



- child - A field instance that should be used for validating the values in the dictionary. If this argument is not provided then values in the mapping will not be validated.

- allow_empty - Designates if empty dictionaries are allowed.

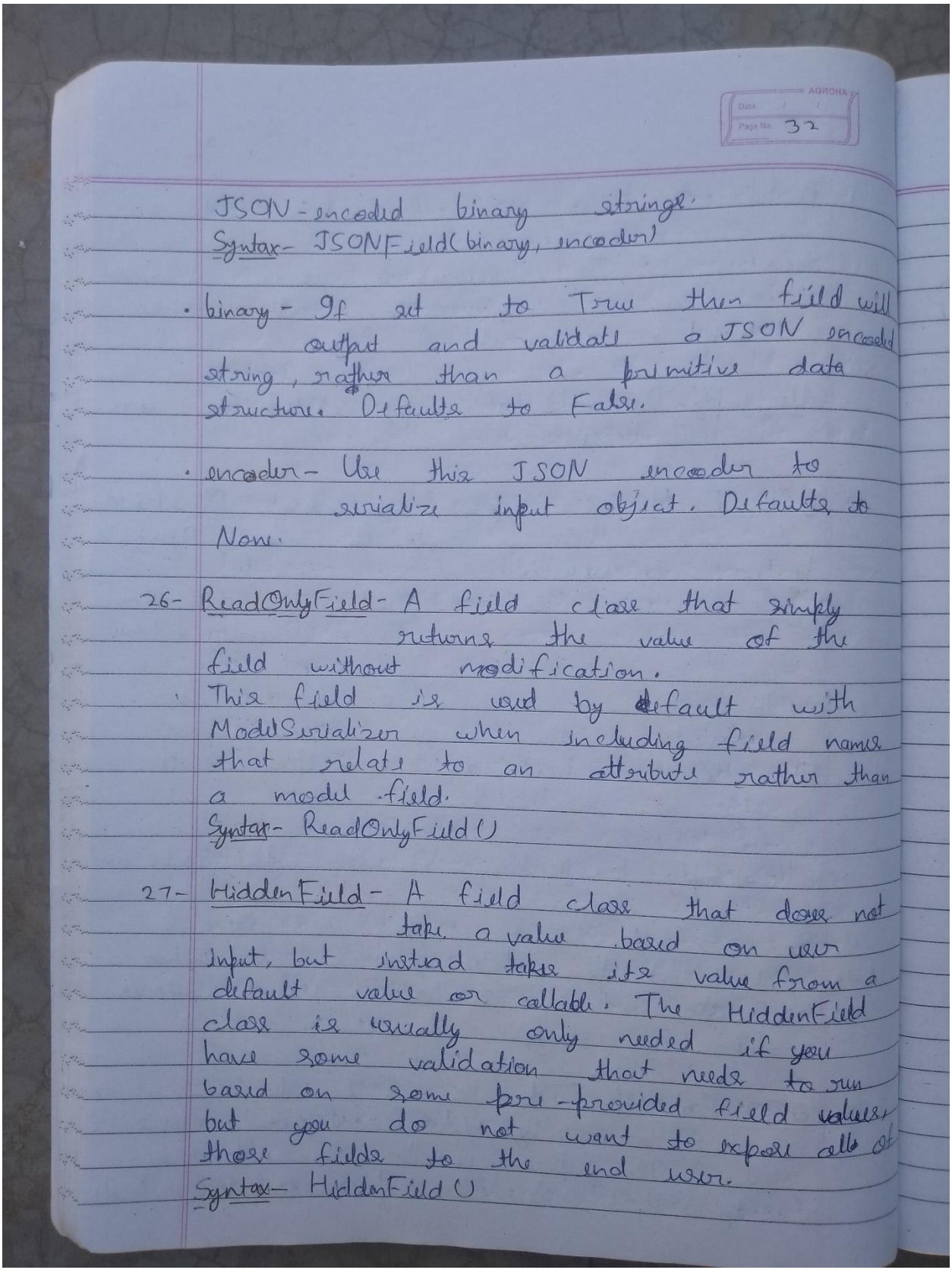
24- HStoreField - A preconfigured DictField that is compatible with Django's postgres HStoreField.
Syntax - HStoreField(child=<A_FIELD_INSTANCE>, allow_empty=True)

- child - A field instance that is used for validating the values in the dictionary. The default child field accepts both empty strings and null values.

- allow_empty - Designates if empty dictionaries are allowed.

Note - Note that the child must be an instance of CharField, as hstore extension stores values as strings.

25- JSONField - A field class that validates that the incoming data structure consists of valid JSON primitives. In its alternate binary mode, it will represent and validate

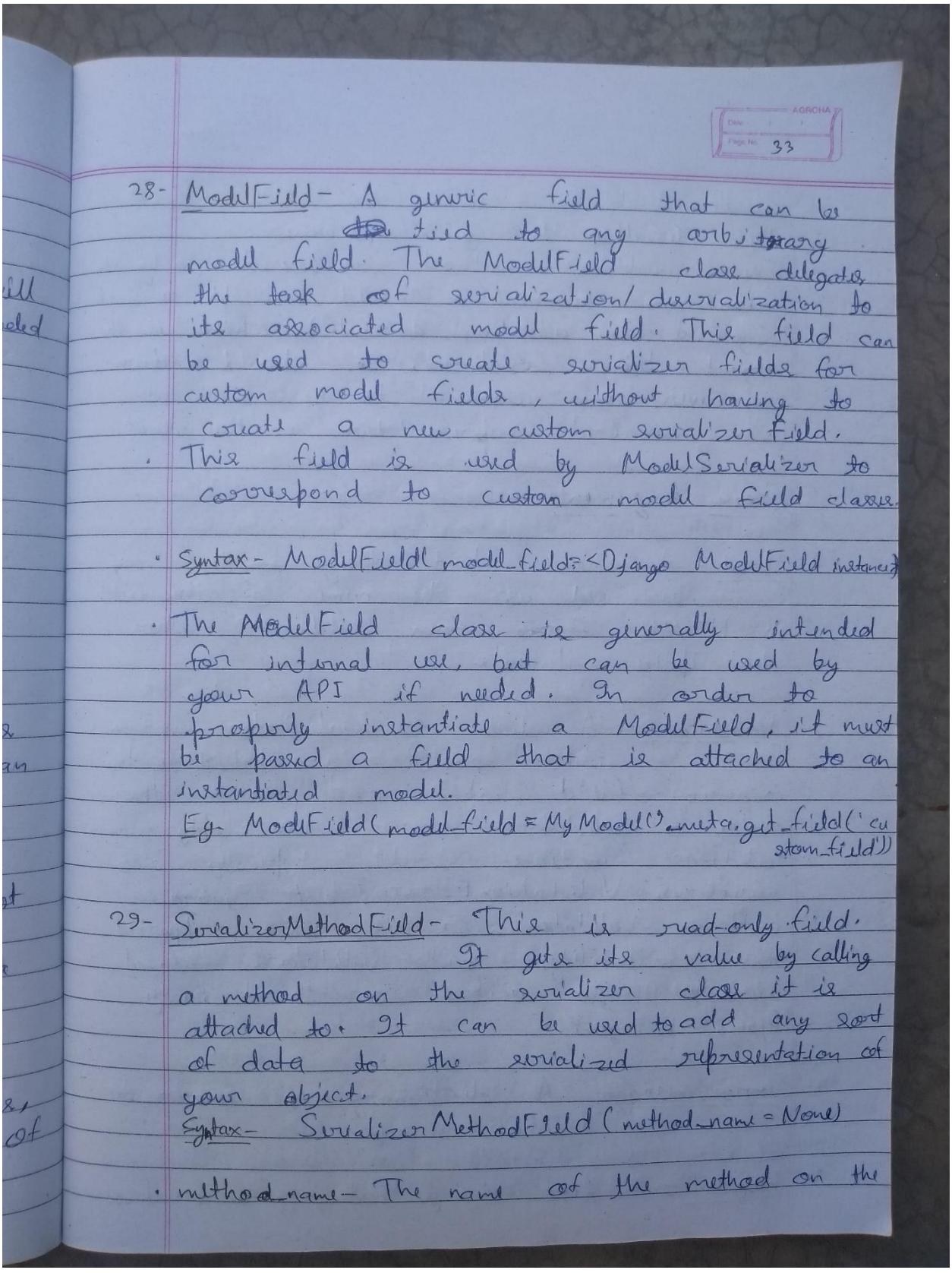


JSON-encoded binary strings.
Syntax- JSONField(binary, encoder)

- binary - If set to True then field will output and validate a JSON encoded string, rather than a primitive data structure. Defaults to False.
- encoder - Use this JSON encoder to serialize input object. Defaults to None.

26- ReadOnlyField - A field class that simply returns the value of the field without modification.
This field is used by default with ModelSerializer when including field names that relate to an attribute rather than a model field.
Syntax- ReadOnlyField()

27- HiddenField - A field class that does not take a value based on user input, but instead takes its value from a default value or callable. The HiddenField class is usually only needed if you have some validation that needs to run based on some pre-provided field values, but you do not want to expose all of those fields to the end user.
Syntax- HiddenField()



28- ModelField - A generic field that can be tied to any arbitrary model field. The ModelField class delegates the task of serialization/deserialization to its associated model field. This field can be used to create serializer fields for custom model fields, without having to create a new custom serializer field.

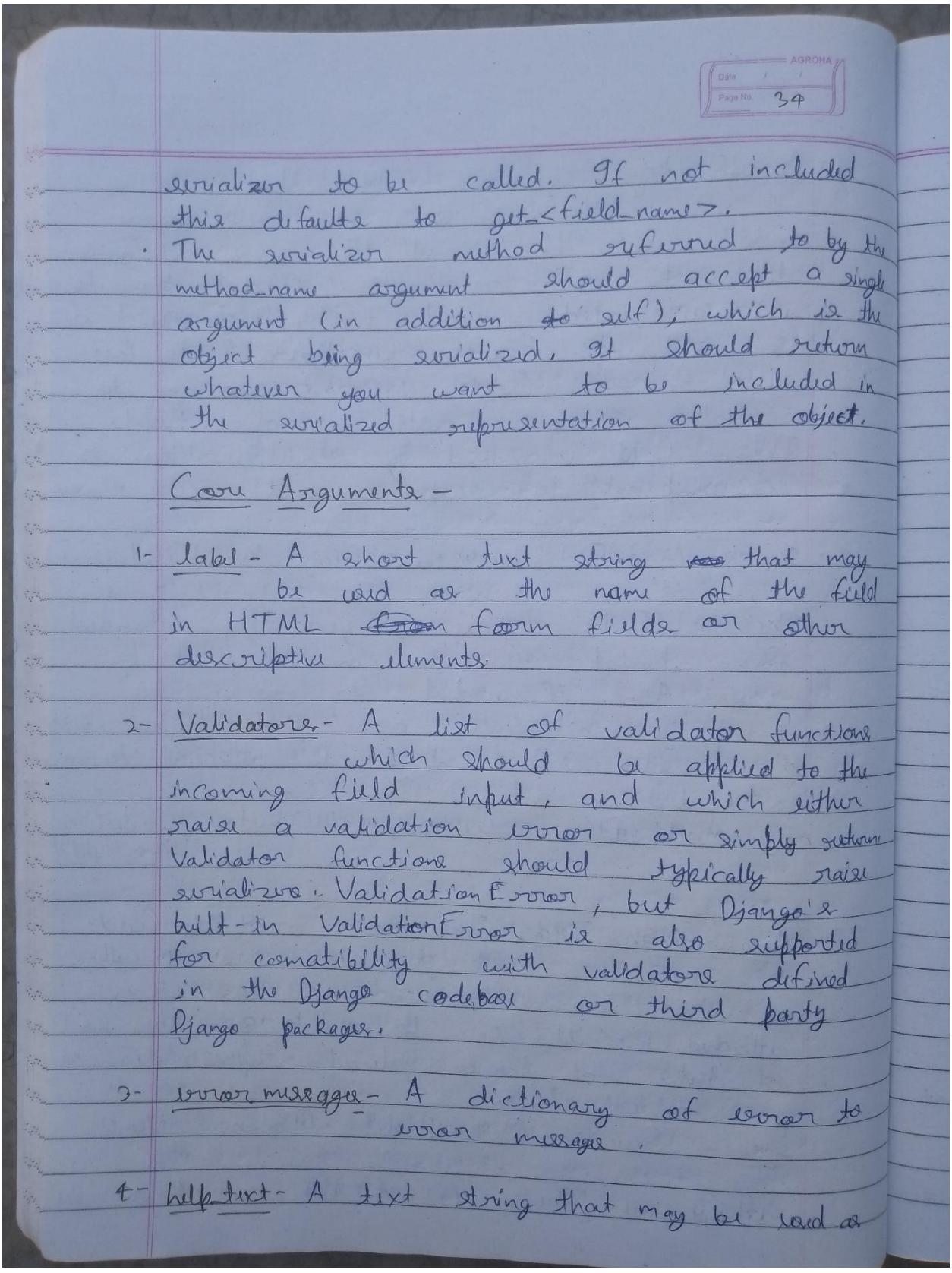
- This field is used by ModelSerializer to correspond to custom model field classes.

- Syntax - ModelField(model_field=<Django ModelField instance>)

- The ModelField class is generally intended for internal use, but can be used by your API if needed. In order to properly instantiate a ModelField, it must be passed a field that is attached to an instantiated model.
  Eg- ModelField(model_field=MyModel()._meta.get_field('custom_field'))

29- SerializerMethodField - This is read-only field. It gets its value by calling a method on the serializer class it is attached to. It can be used to add any sort of data to the serialized representation of your object.
Syntax - SerializerMethodField(method_name=None)

- method_name - The name of the method on the



serializer to be called. If not included this defaults to get_<field_name>.
- The serializer method referred to by the method_name argument should accept a single argument (in addition to self), which is the object being serialized. It should return whatever you want to be included in the serialized representation of the object.

Core Arguments -

1- label - A short text string that may be used as the name of the field in HTML form fields or other descriptive elements.

2- Validators - A list of validator functions which should be applied to the incoming field input, and which either raise a validation error or simply return. Validator functions should typically raise serializers.ValidationError, but Django's built-in ValidationError is also supported for compatibility with validators defined in the Django codebase or third party Django packages.

3- error_messages - A dictionary of error to error messages.

4- help_text - A text string that may be used as

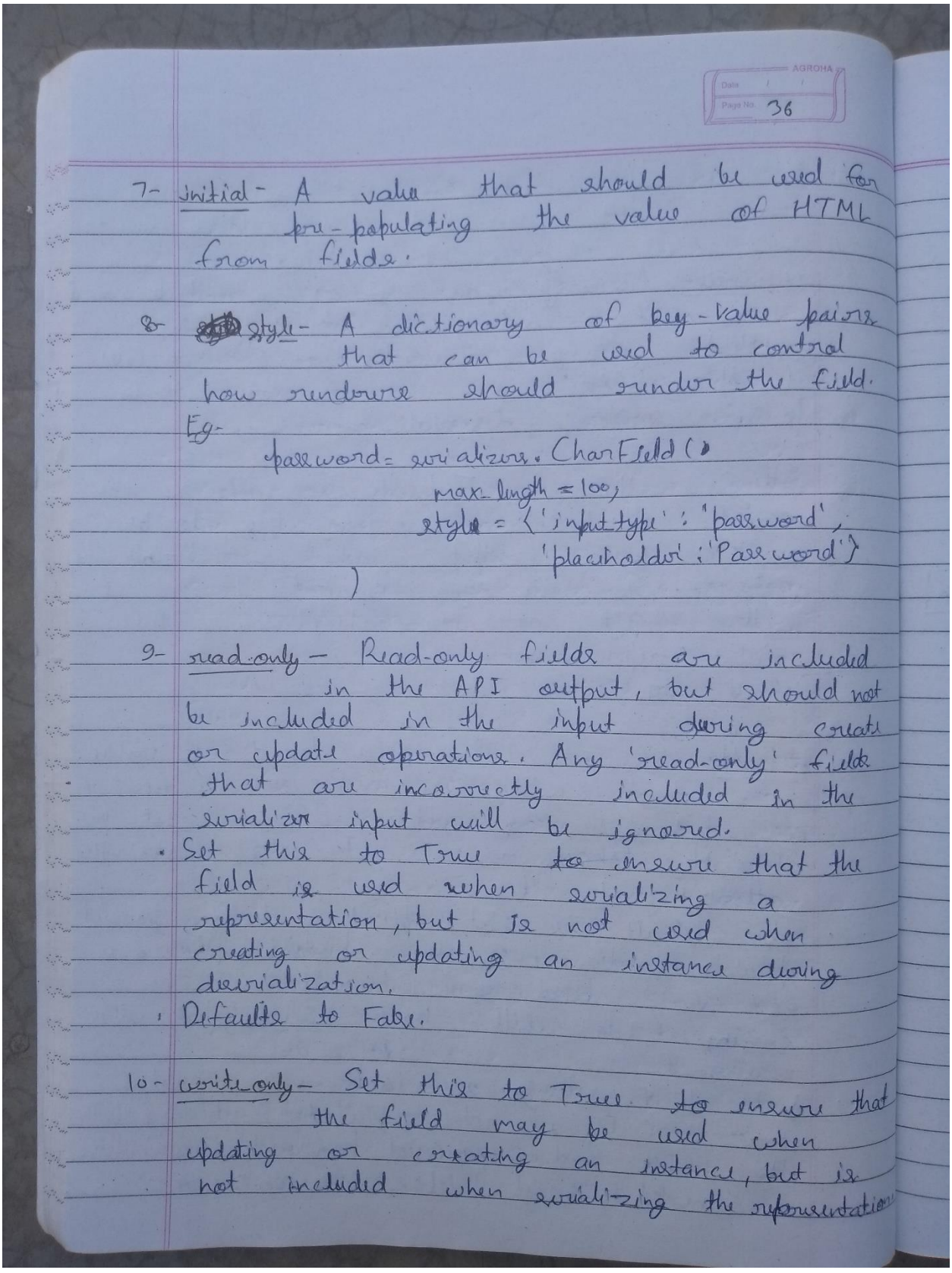
a description of the field in HTML form field or other descriptive elements.

5- required - Normally an error will be raised if a field is not ~~supported~~ supplied during deserialization. Set to false if this field is not required to be present during deserialization.

- Setting this to False also allows the object attribute or dictionary key to be omitted from output when serializing the instance. If the key is not present it will simply not be included in the output representation.
- Default to True.

6- default - If set, this gives the default value that will be used for the field if no input value is supplied. If not set the default behaviour is to not populate the attribute at all.

- The default is not applied during partial update operations. In the partial update case only fields that are provided in the coming data will have a validated value returned.
- Note that setting a default value implies that the field is not required. Including both the default and required keyword arguments is invalid and will raise an error.



7- initial - A value that should be used for pre-populating the value of HTML form fields.

8- style - A dictionary of key-value pairs that can be used to control how renderer should render the field.

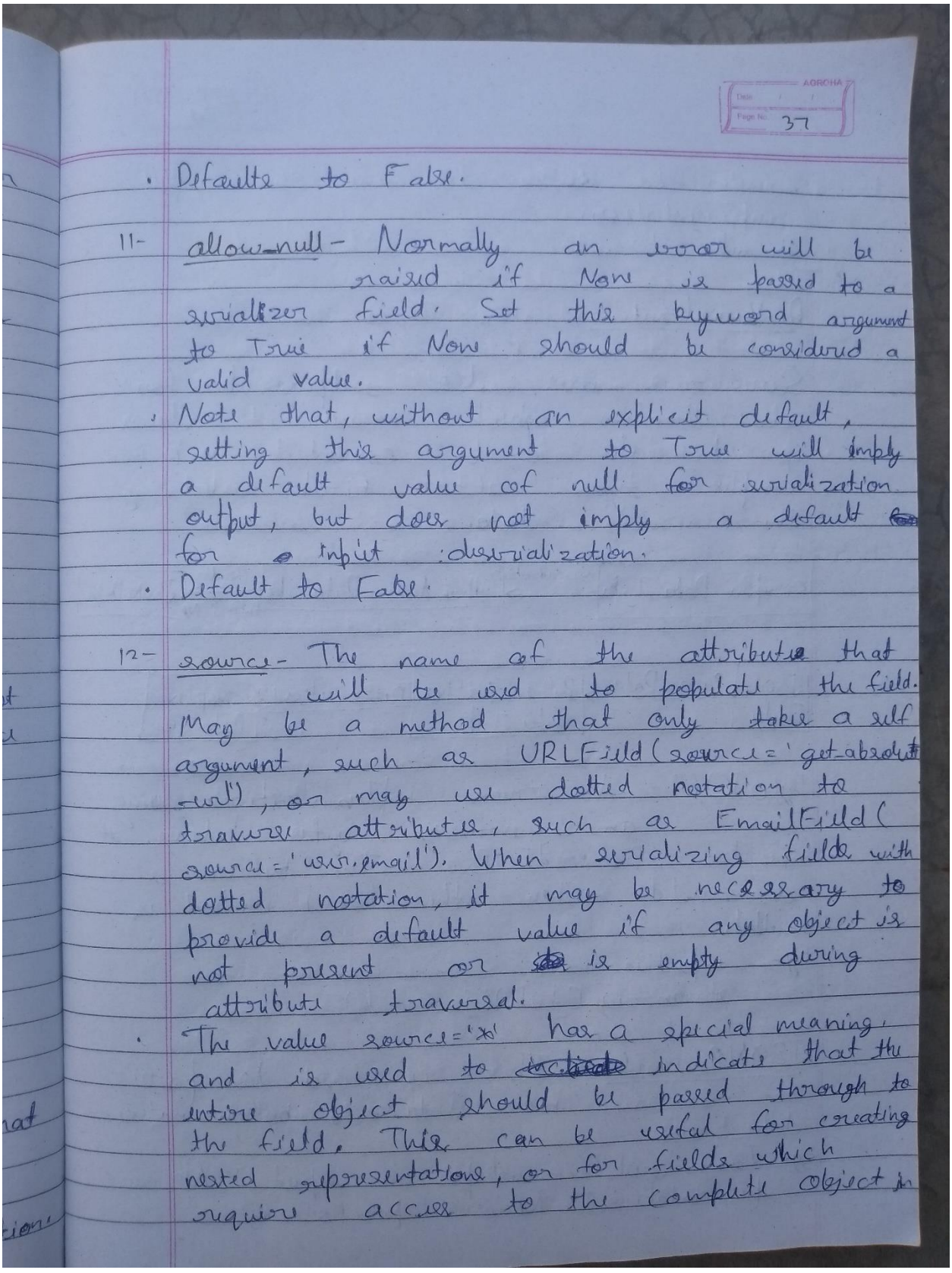
Eg-

```
password = serializers.CharField(
    max_length = 100,
    style = {'input_type': 'password',
             'placeholder': 'Password'}
)
```

9- read_only - Read-only fields are included in the API output, but should not be included in the input during create or update operations. Any 'read-only' fields that are incorrectly included in the serializer input will be ignored.

- Set this to True to ensure that the field is used when serializing a representation, but is not used when creating or updating an instance during deserialization.
- Defaults to False.

10- write_only - Set this to True to ensure that the field may be used when updating or creating an instance, but is not included when serializing the representation.



- Defaults to False.

11- allow_null - Normally an error will be raised if None is passed to a serializer field. Set this keyword argument to True if None should be considered a valid value.
- Note that, without an explicit default, setting this argument to True will imply a default value of null for serialization output, but does not imply a default for input deserialization.
- Default to False.

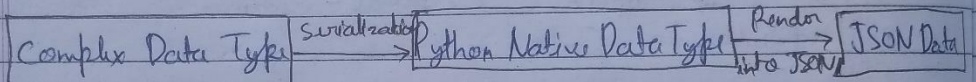
12- source - The name of the attribute that will be used to populate the field. May be a method that only takes a self argument, such as URLField(source='get_absolute_url'), or may use dotted notation to traverse attributes, such as EmailField(source='user.email'). When serializing fields with dotted notation, it may be necessary to provide a default value if any object is not present or is empty during attribute traversal.
- The value source='*' has a special meaning, and is used to indicate that the entire object should be passed through to the field. This can be useful for creating nested representations, or for fields which require access to the complete object in

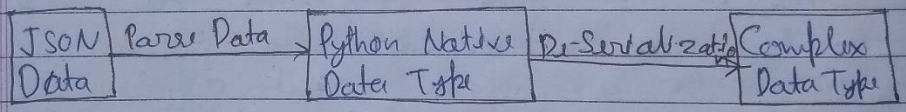
order to ~~doctor~~ determine the output representation.

8. Defaults to the name of the field.

De-Serialization -

Serializers are also responsible for deserialization which means it allows parsed data to be converted back into complex types, after first validating the incoming data.



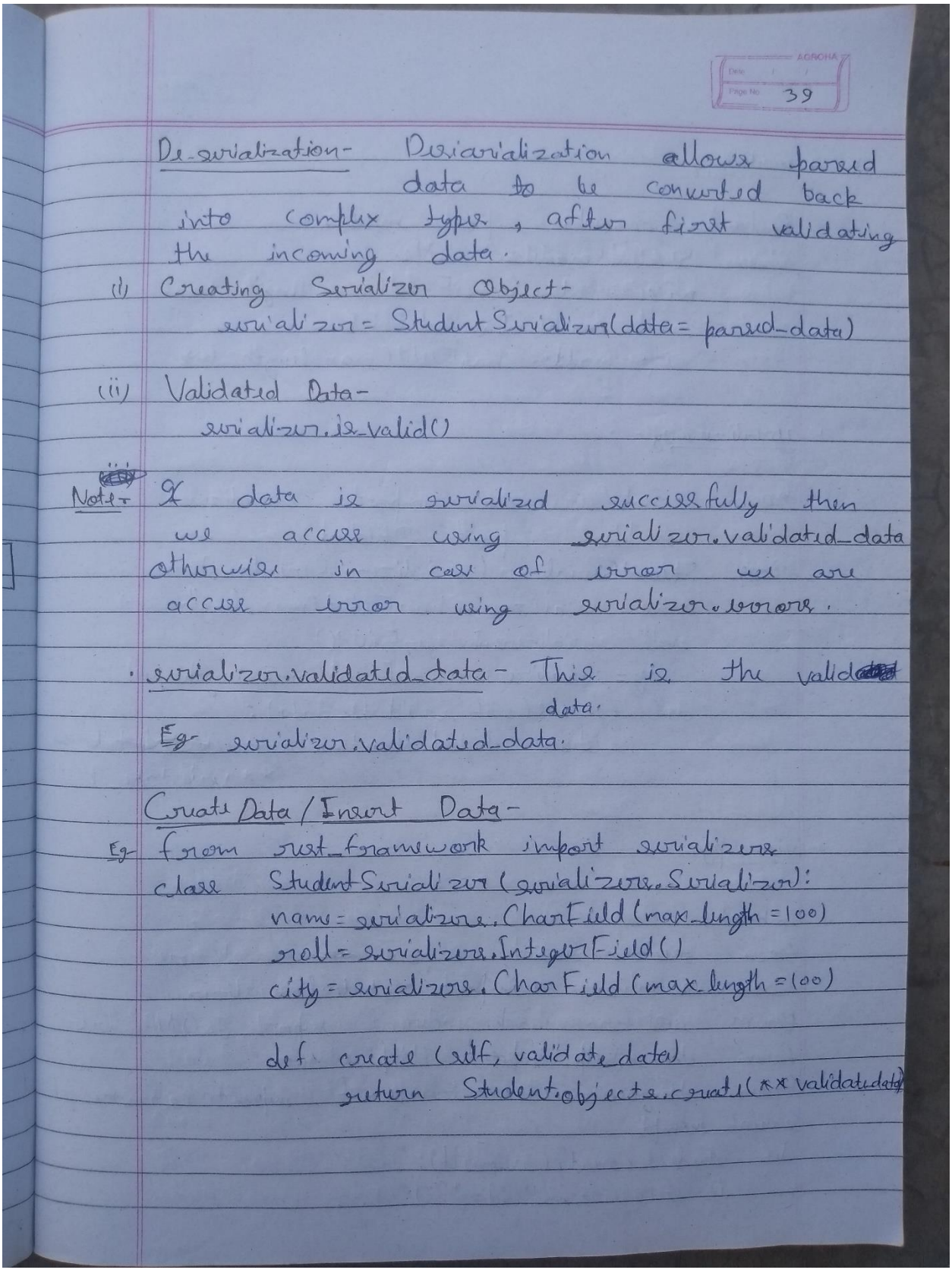


BytesIO() - A stream implementation using an in-memory bytes buffer. It inherits BufferedIOBase. The buffer is discarded when the close() method is called.

Eg- import io
stream = io.BytesIO(json_data)

JSONParser() - This is used to parse json data to python native data type.

Eg- from rest_framework.parsers import JSONParser
parsed_data = JSONParser().parse(stream)



Page No 39

De-serialization- Deserialization allows parsed data to be converted back into complex type, after first validating the incoming data.

(i) Creating Serializer Object-
serializer = StudentSerializer(data=parsed_data)

(ii) Validated Data-
serializer.is_valid()

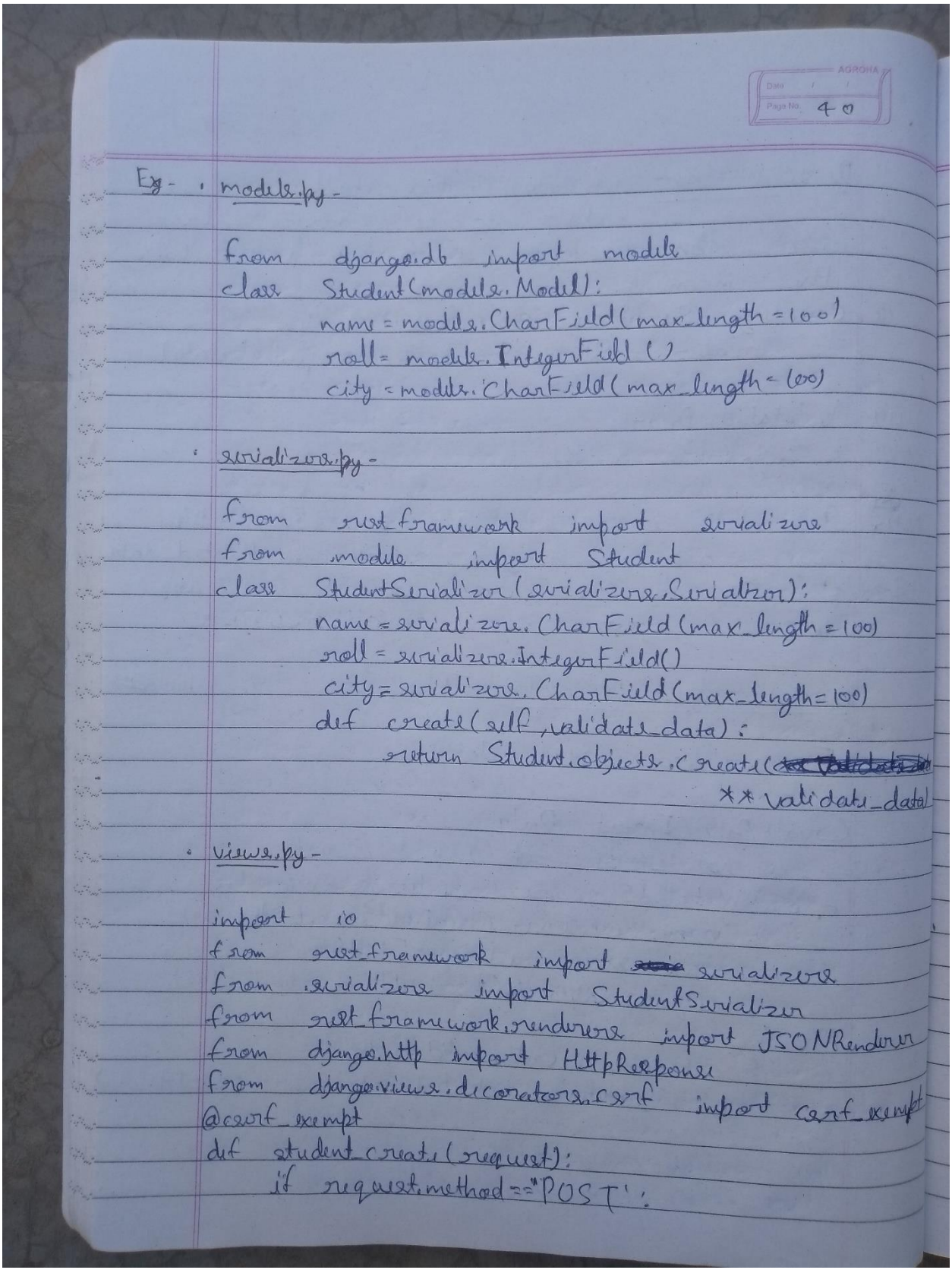
Note:- If data is serialized successfully then we access using serializer.validated_data otherwise in case of error we are access error using serializer.errors.

- serializer.validated_data - This is the validated data.

Eg- serializer.validated_data.

Create Data / Insert Data-

Eg-
```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)

    def create(self, validated_data)
        return Student.objects.create(**validated_data)
```



Ex- • models.py -

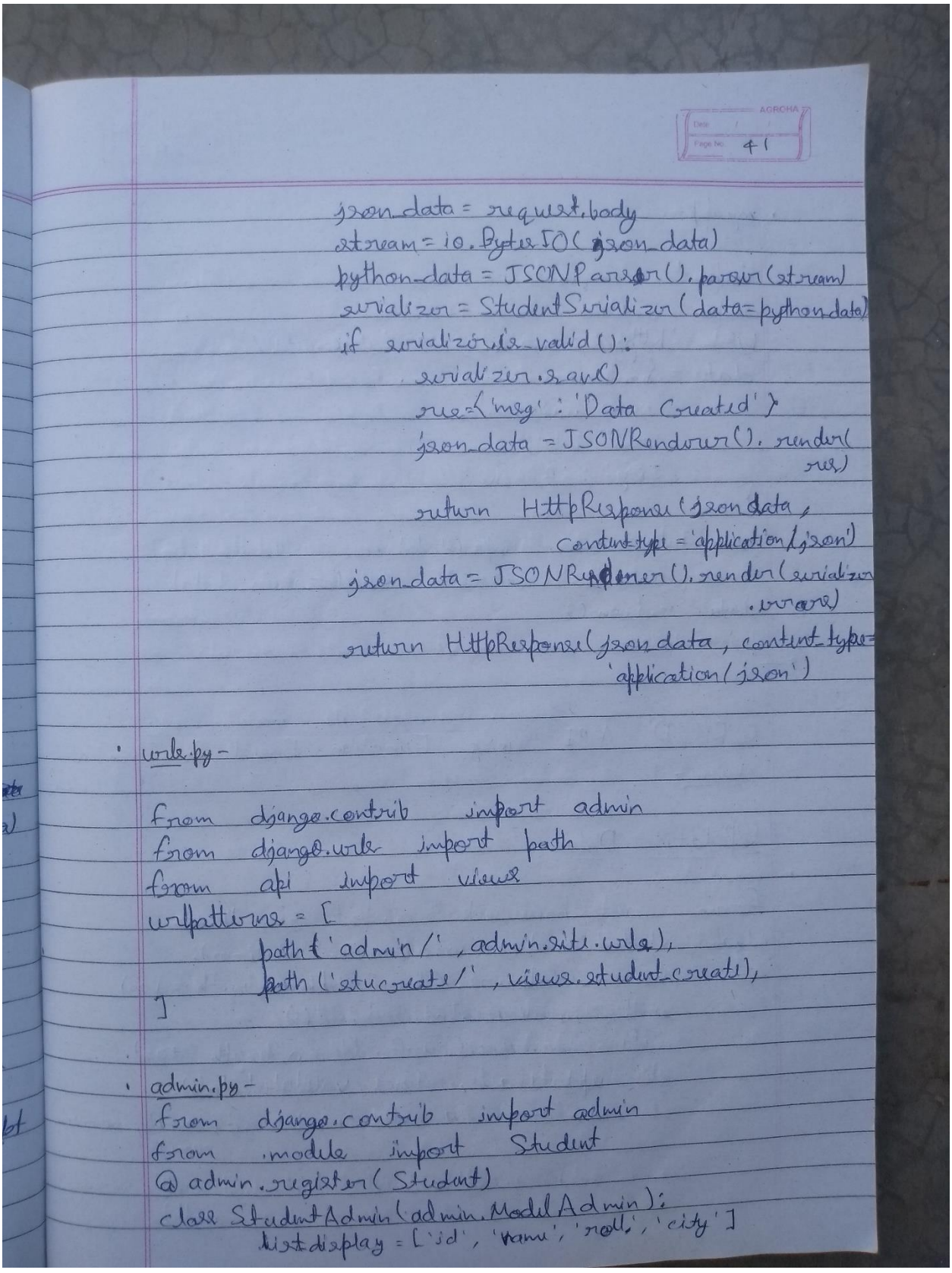
```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=100)
    roll = models.IntegerField()
    city = models.CharField(max_length=100)
```

• serializers.py -

```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
    def create(self, validated_data):
        return Student.objects.create(**validated_data)
```

• views.py -

```
import io
from rest_framework import serializers
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
@csrf_exempt
def student_create(request):
    if request.method=='POST':
```



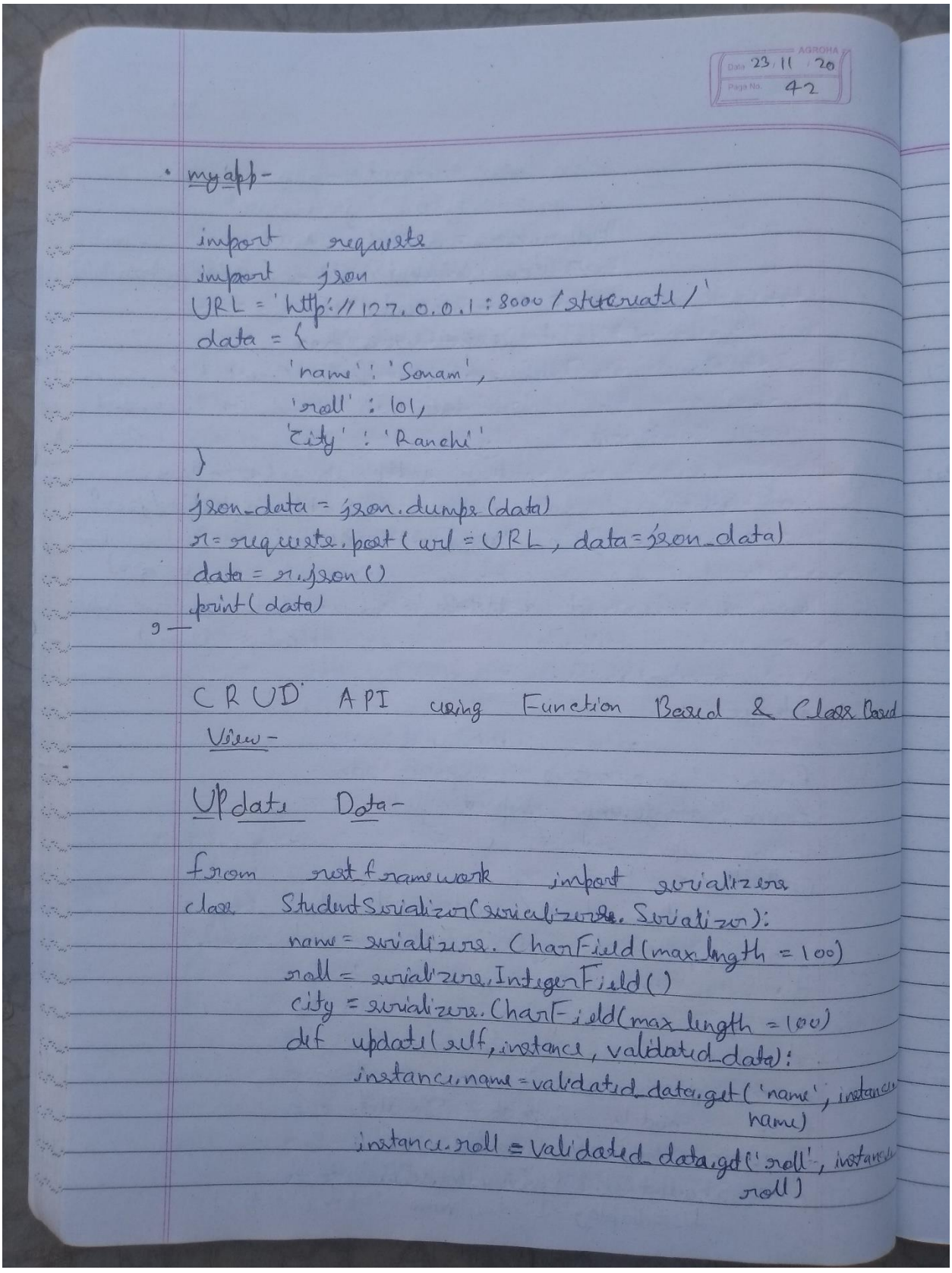
```
json_data = request.body
stream = io.BytesIO(json_data)
python_data = JSONParser().parse(stream)
serializer = StudentSerializer(data=python_data)
if serializer.is_valid():
    serializer.save()
    res={'msg' : 'Data Created'}
    json_data = JSONRenderer().render(
                                res)

    return HttpResponse(json_data,
                content_type = 'application/json')
json_data = JSONRenderer().render(serializer
                                .errors)

return HttpResponse(json_data, content_type=
                    'application/json')
```

- urls.py -

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
        path('admin/', admin.site.urls),
        path('stucreate/', views.student_create),
]
```

- admin.py -

```
from django.contrib import admin
from .module import Student
@admin.register(Student)
class StudentAdmin(admin.ModelAdmin):
        list_display = ['id', 'name', 'roll', 'city']
```



- myapp-

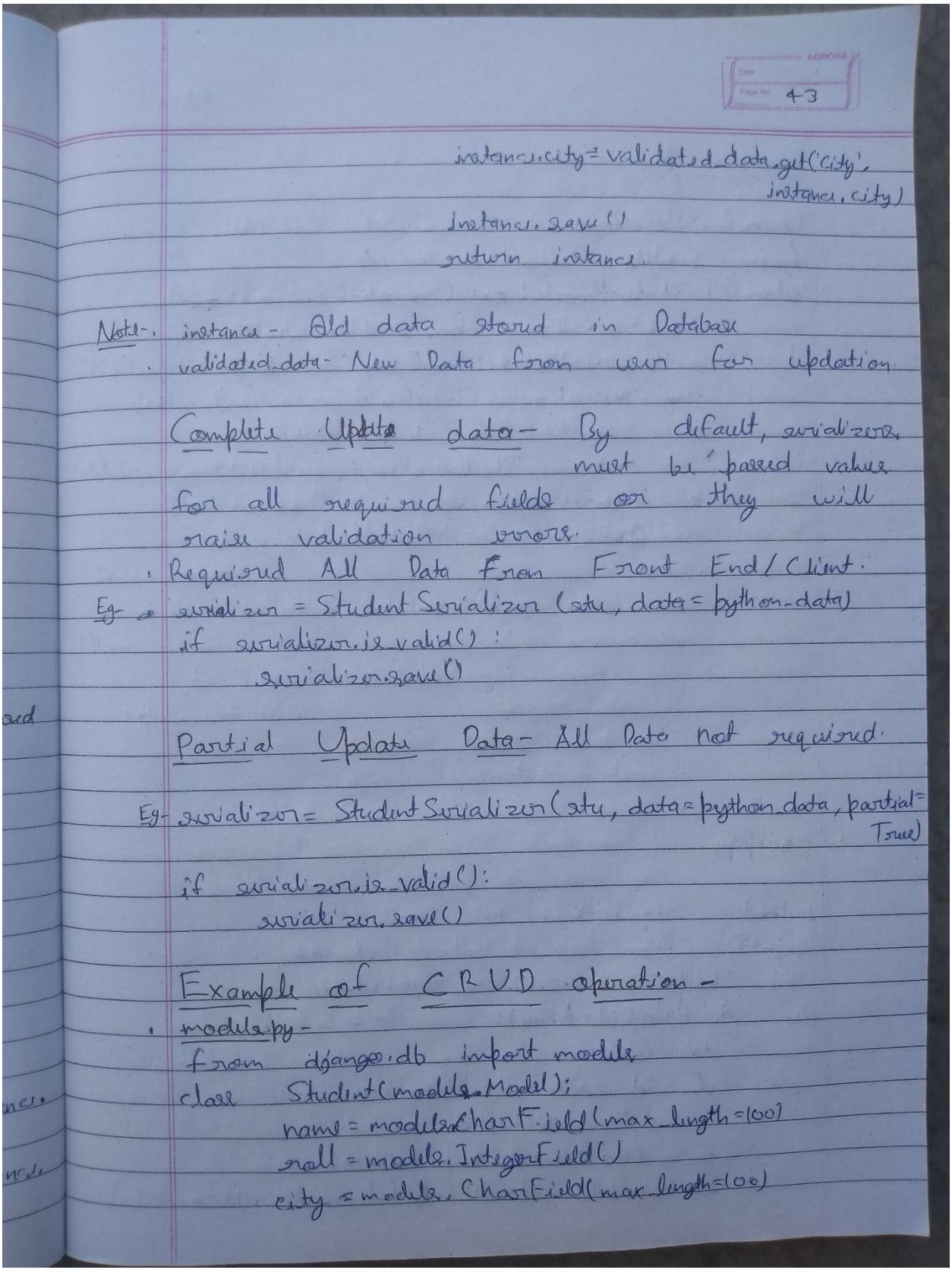
```
import requests
import json
URL = 'http://127.0.0.1:8000/studentapi/'
data = {
        'name' : 'Sonam',
        'roll' : 101,
        'city' : 'Ranchi'
}
json_data = json.dumps(data)
r = requests.post(url = URL, data = json_data)
data = r.json()
print(data)
```

9—

CRUD API using Function Based & Class Based View-

Update Data-

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length = 100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length = 100)
    def update(self, instance, validated_data):
        instance.name = validated_data.get('name', instance.
                                           name)
        instance.roll = validated_data.get('roll', instance.
                                           roll)
```



instance.city = validated_data.get('city', instance.city)
instance.save()
return instance.

Note:- instance - Old data stored in Database
validated_data - New Data from user for updation.

Complete Update data - By default, serializers must be passed values for all required fields or they will raise validation errors.
Required All Data From Front End / Client.

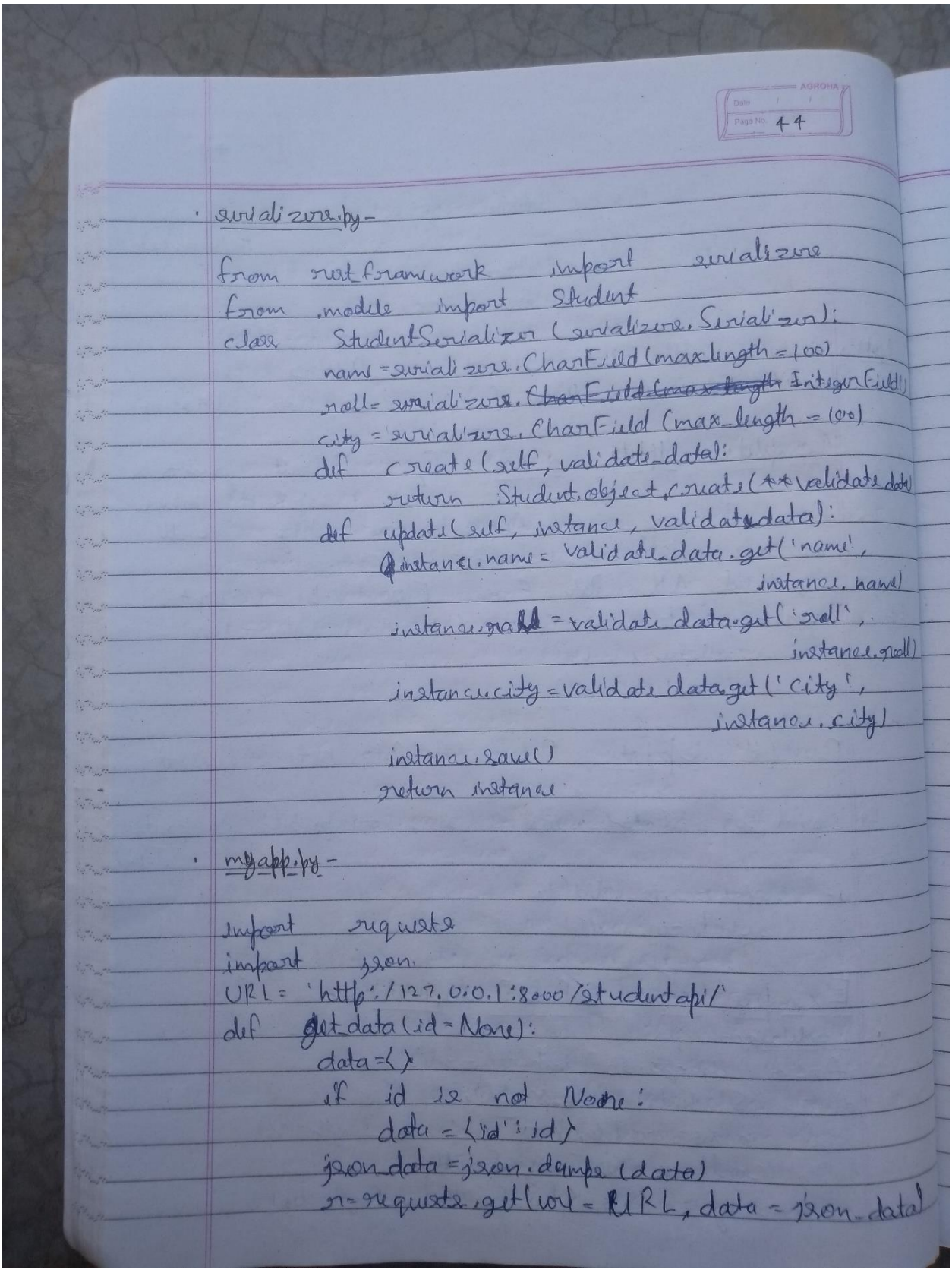
Eg - serializer = StudentSerializer(stu, data=python_data)
if serializer.is_valid():
serializer.save()

Partial Update Data - All Data not required.

Eg - serializer = StudentSerializer(stu, data=python_data, partial=True)
if serializer.is_valid():
serializer.save()

Example of CRUD operation -

- models.py -
from django.db import models
class Student(models.Model):
name = models.CharField(max_length=100)
roll = models.IntegerField()
city = models.CharField(max_length=100)

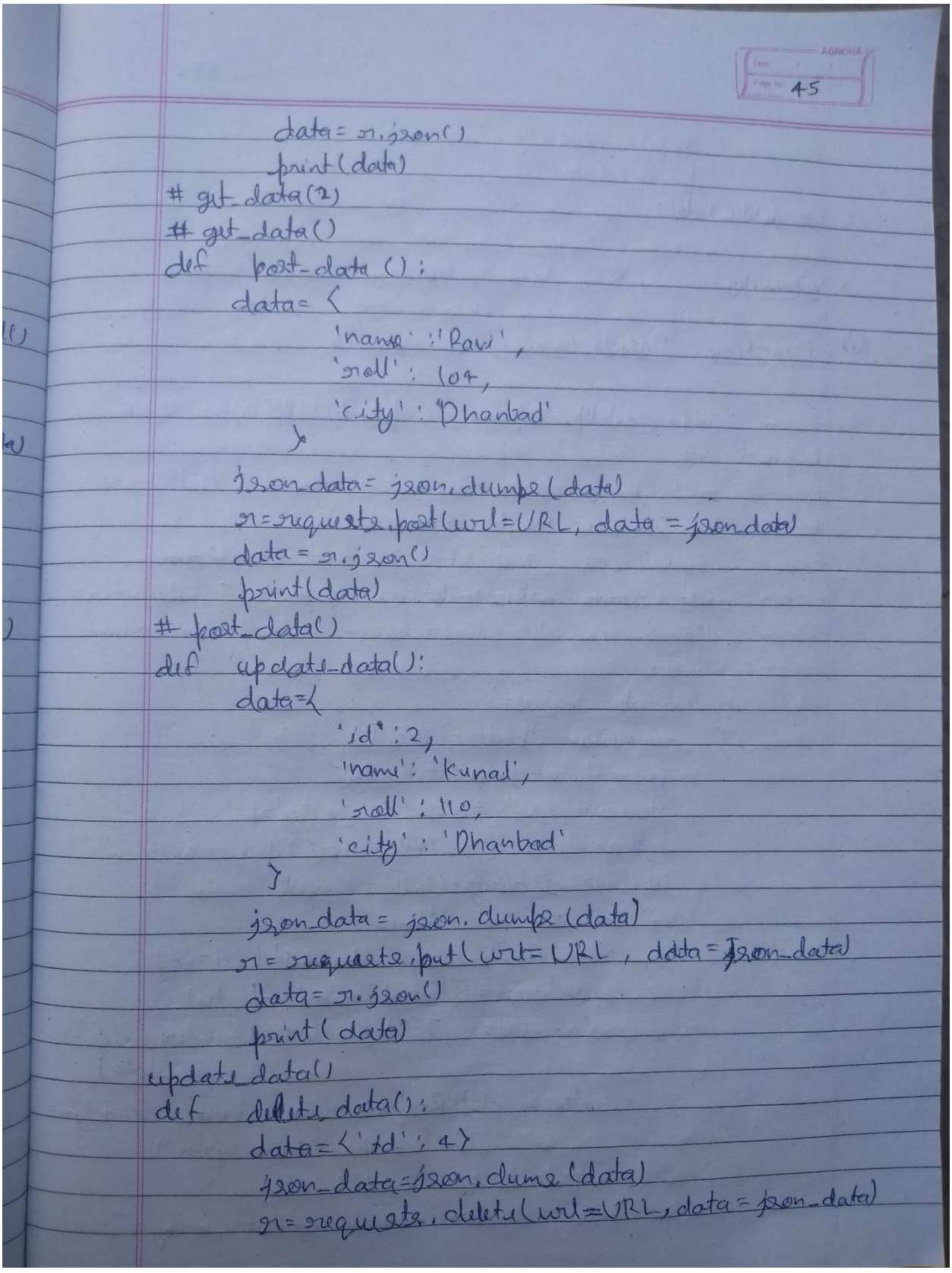


- serializers.py -

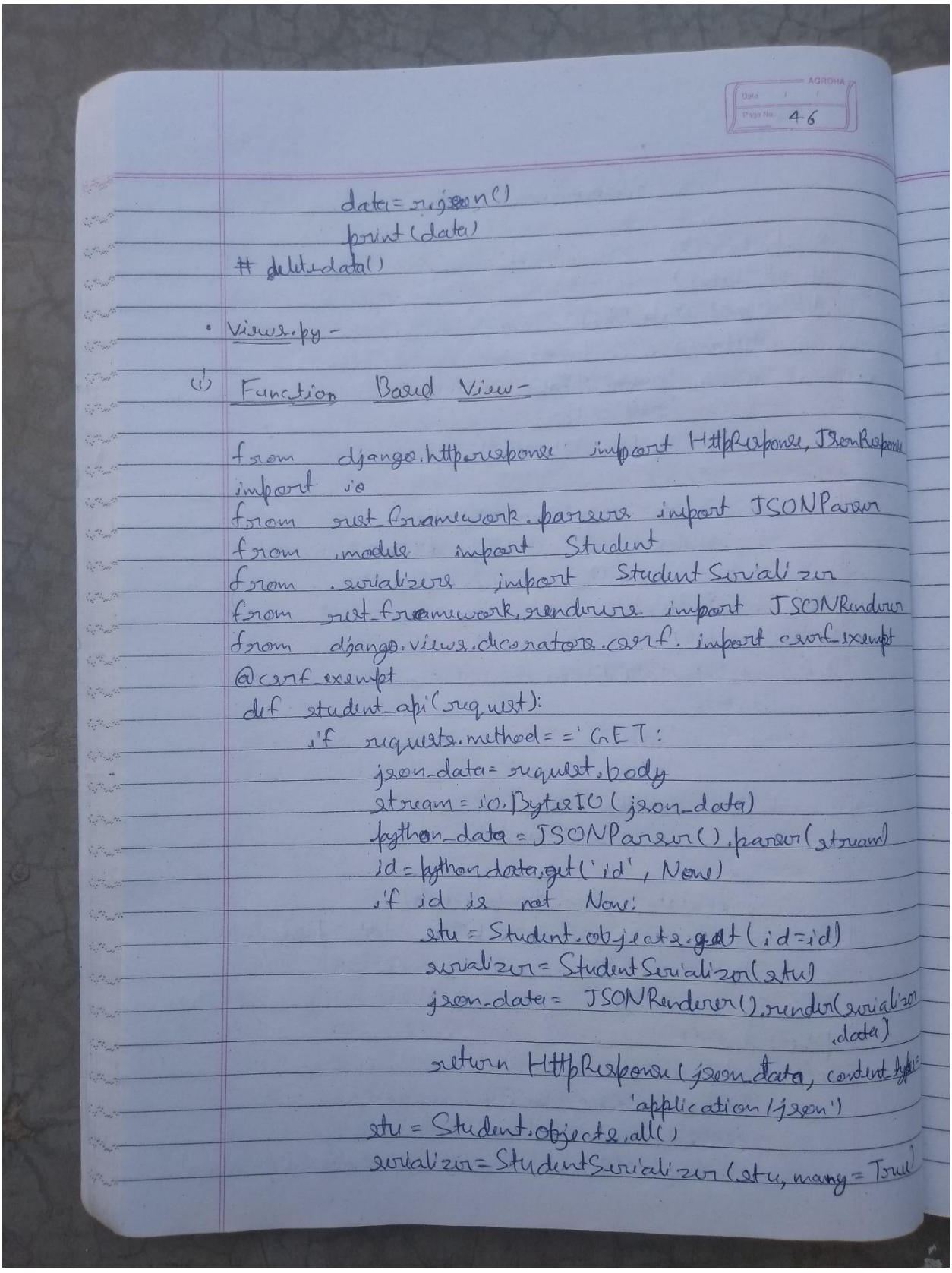
```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
    def create(self, validated_data):
        return Student.objects.create(**validated_data)
    def update(self, instance, validated_data):
        instance.name = validated_data.get('name',
                                           instance.name)
        instance.roll = validated_data.get('roll',
                                           instance.roll)
        instance.city = validated_data.get('city',
                                           instance.city)
        instance.save()
        return instance
```

- myapp.py -

```
import requests
import json
URL = 'http://127.0.0.1:8000/studentapi/'
def get_data(id=None):
    data={}
    if id is not None:
        data = {'id': id}
    json_data = json.dumps(data)
    r = requests.get(url = URL, data = json_data)
```



```
        data = r.json()
        print(data)
# get_data(2)
# get_data()
def post_data():
    data = {
            'name' : 'Ravi',
            'roll' : 104,
            'city' : 'Dhanbad'
        }
    json_data = json.dumps(data)
    r = requests.post(url=URL, data = json_data)
    data = r.json()
    print(data)
# post_data()
def update_data():
    data={
            'id' : 2,
            'name' : 'Kunal',
            'roll' : 110,
            'city' : 'Dhanbad'
        }
    json_data = json.dumps(data)
    r = requests.put(url=URL, data=json_data)
    data = r.json()
    print(data)
update_data()
def delete_data():
    data = {'id' : 4}
    json_data = json.dumps(data)
    r = requests.delete(url=URL, data = json_data)
```

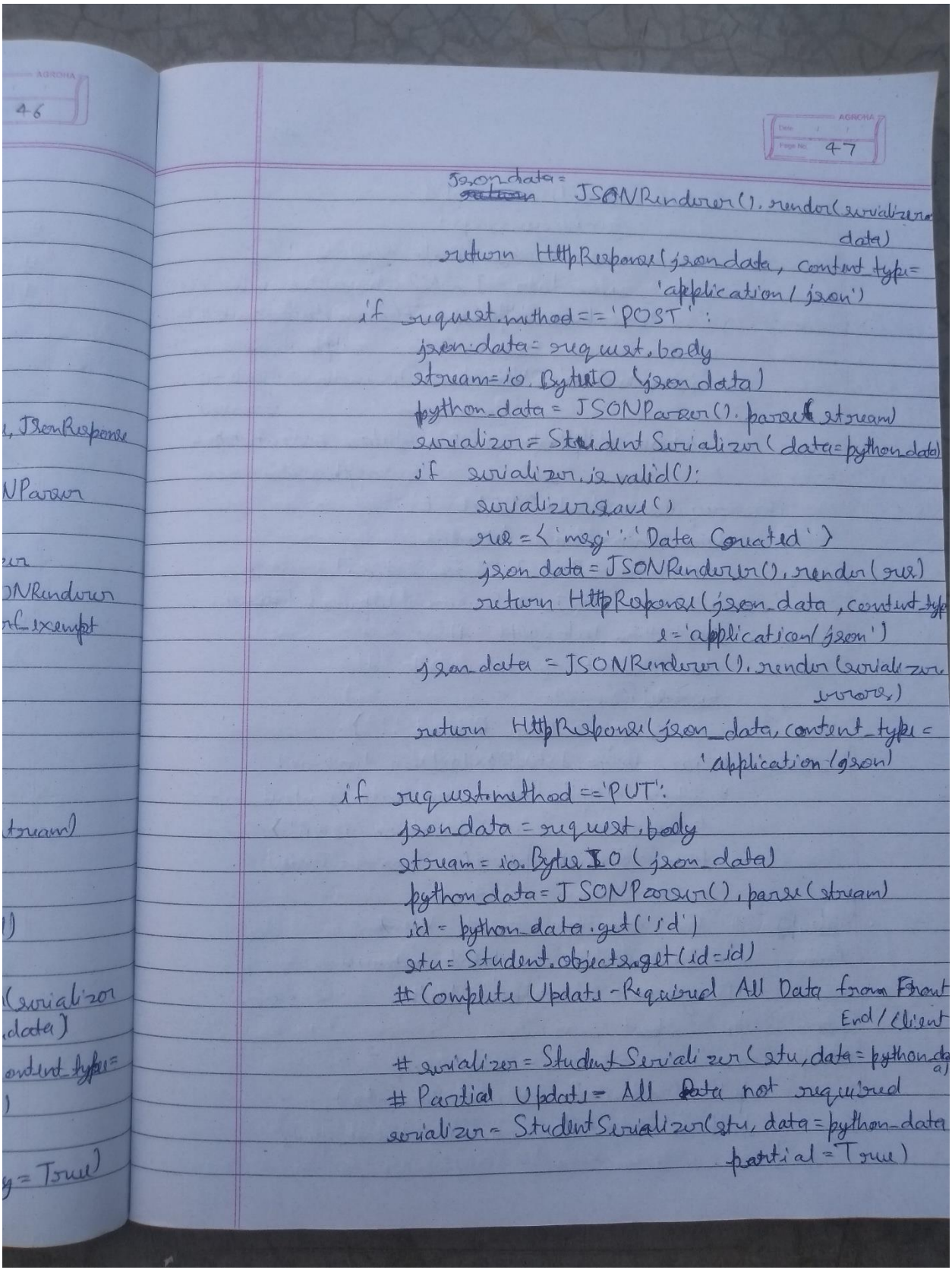


```
        data = r.json()
        print(data)
# deletedata()
```

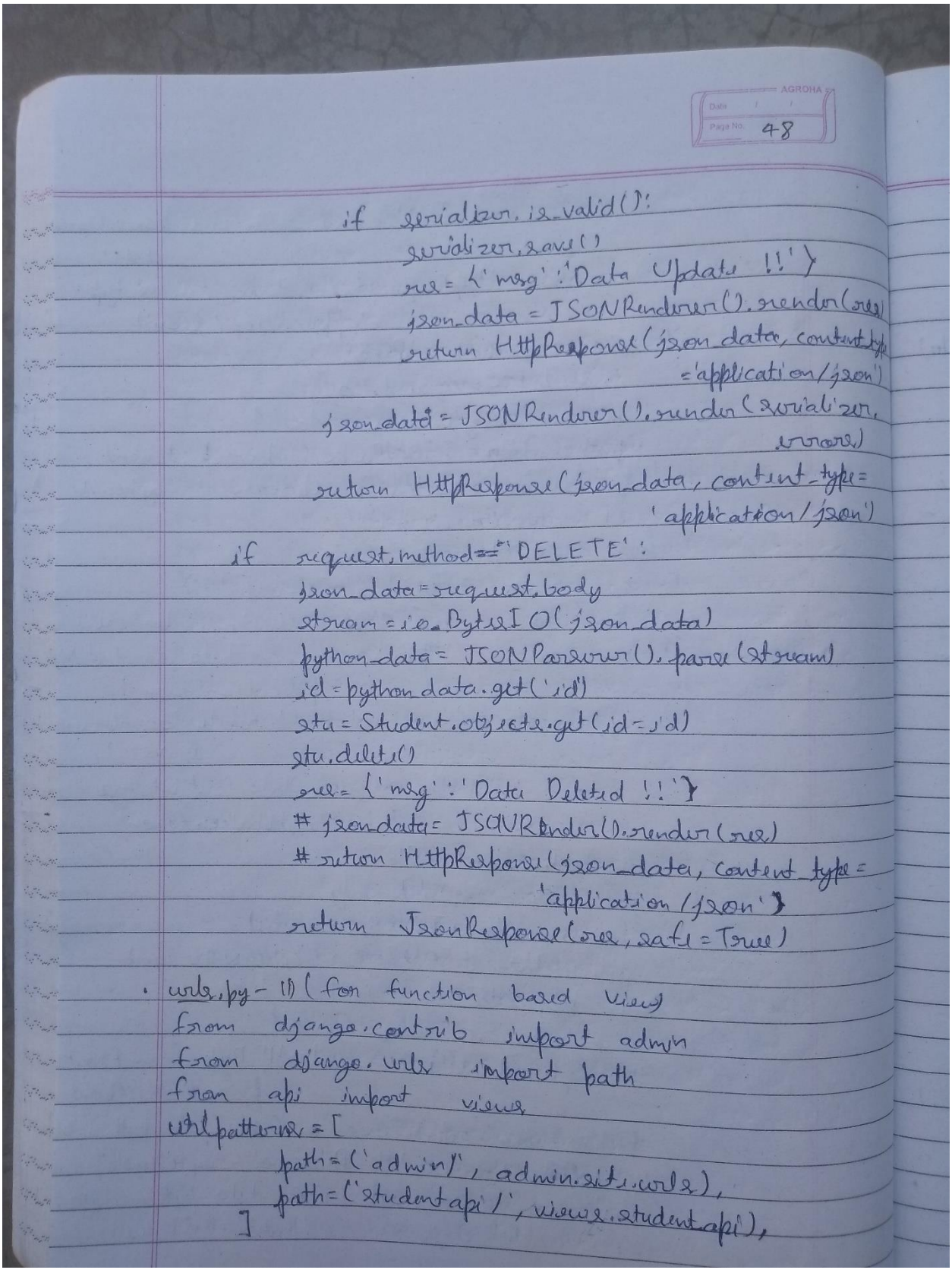
- views.py -

(i) Function Based View -

```
from django.http.response import HttpResponse, JsonResponse
import io
from rest_framework.parsers import JSONParser
from .models import Student
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.views.decorators.csrf import csrf_exempt
@csrf_exempt
def student_api(request):
    if request.method == 'GET':
        json_data = request.body
        stream = io.BytesIO(json_data)
        python_data = JSONParser().parse(stream)
        id = python_data.get('id', None)
        if id is not None:
            stu = Student.objects.get(id=id)
            serializer = StudentSerializer(stu)
            json_data = JSONRenderer().render(serializer.data)
            return HttpResponse(json_data, content_type='application/json')
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many = True)
```



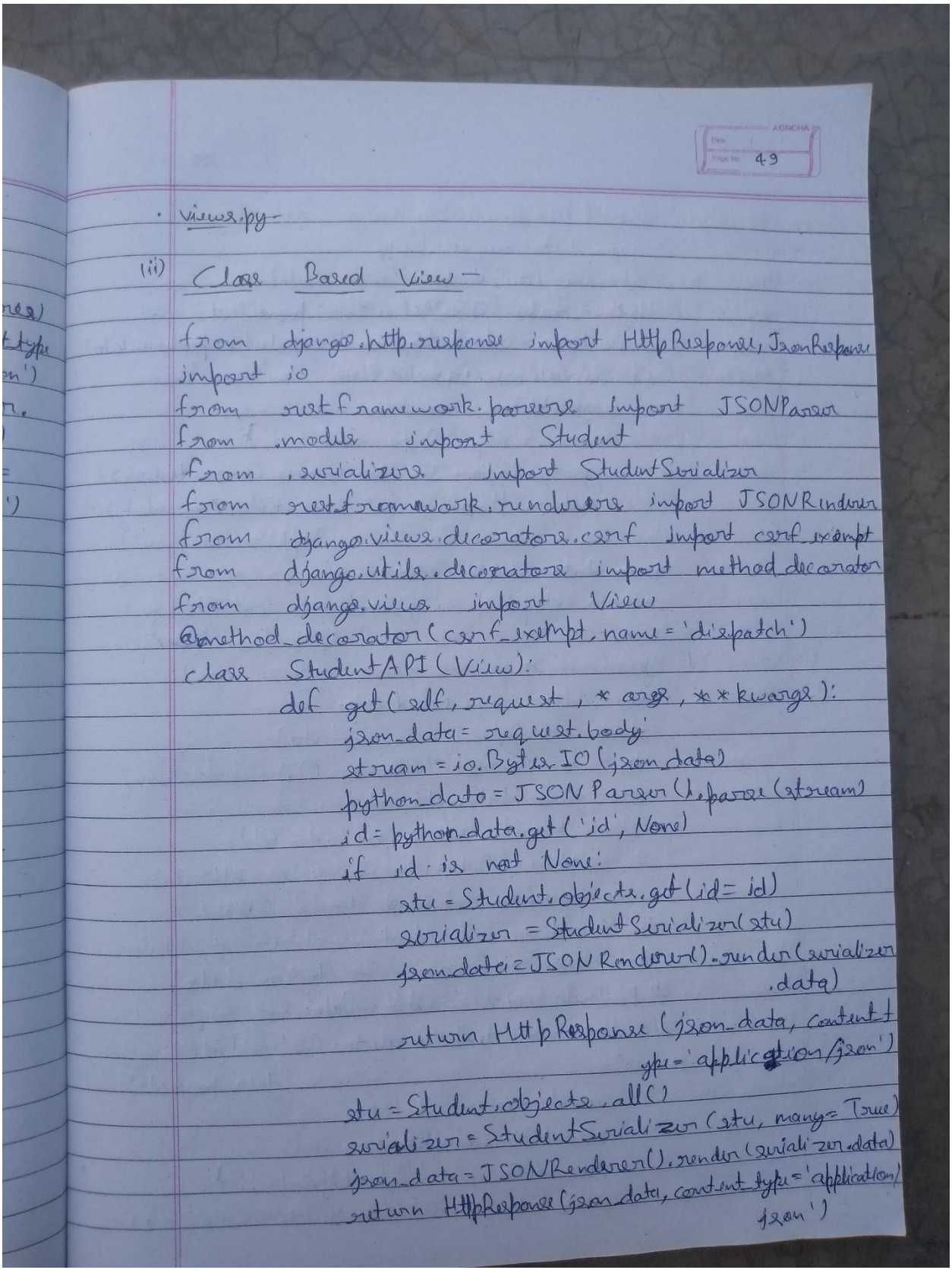
```
                    json_data = JSONRenderer().render(serializer.
                                                      data)
                    return HttpResponse(json_data, content_type=
                                        'application/json')
        if request.method == 'POST':
            json_data = request.body
            stream = io.BytesIO(json_data)
            python_data = JSONParser().parse(stream)
            serializer = StudentSerializer(data=python_data)
            if serializer.is_valid():
                serializer.save()
                res = {'msg': 'Data Created'}
                json_data = JSONRenderer().render(res)
                return HttpResponse(json_data, content_type
                                    ='application/json')
            json_data = JSONRenderer().render(serializer.
                                              errors)
            return HttpResponse(json_data, content_type=
                                'application/json')
        if requestmethod == 'PUT':
            json_data = request.body
            stream = io.BytesIO(json_data)
            python_data = JSONParser().parse(stream)
            id = python_data.get('id')
            stu = Student.objects.get(id=id)
            # Complete Update - Required All Data from Front
                                                    End/Client
            # serializer = StudentSerializer(stu, data=python_da
                                                          ta)
            # Partial Update = All Data not required
            serializer = StudentSerializer(stu, data=python_data
                                           partial=True)
```



```
        if serializer.is_valid():
            serializer.save()
            res = {'msg':'Data Updated !!'}
            json_data = JSONRenderer().render(res)
            return HttpResponse(json_data, content_type
                                =application/json')
        json_data = JSONRenderer().render(serializer.
                                errors)
        return HttpResponse(json_data, content_type=
                                'application/json')
    if request.method=='DELETE':
        json_data=request.body
        stream=io.BytesIO(json_data)
        python_data=JSONParser().parse(stream)
        id=python_data.get('id')
        stu=Student.objects.get(id=id)
        stu.delete()
        res={'msg':'Data Deleted !!'}
        # json_data=JSONRender().render(res)
        # return HttpResponse(json_data, content_type=
                                'application/json')
        return JsonResponse(res, safe=True)
```

- urls.py - 1) (for function based view)

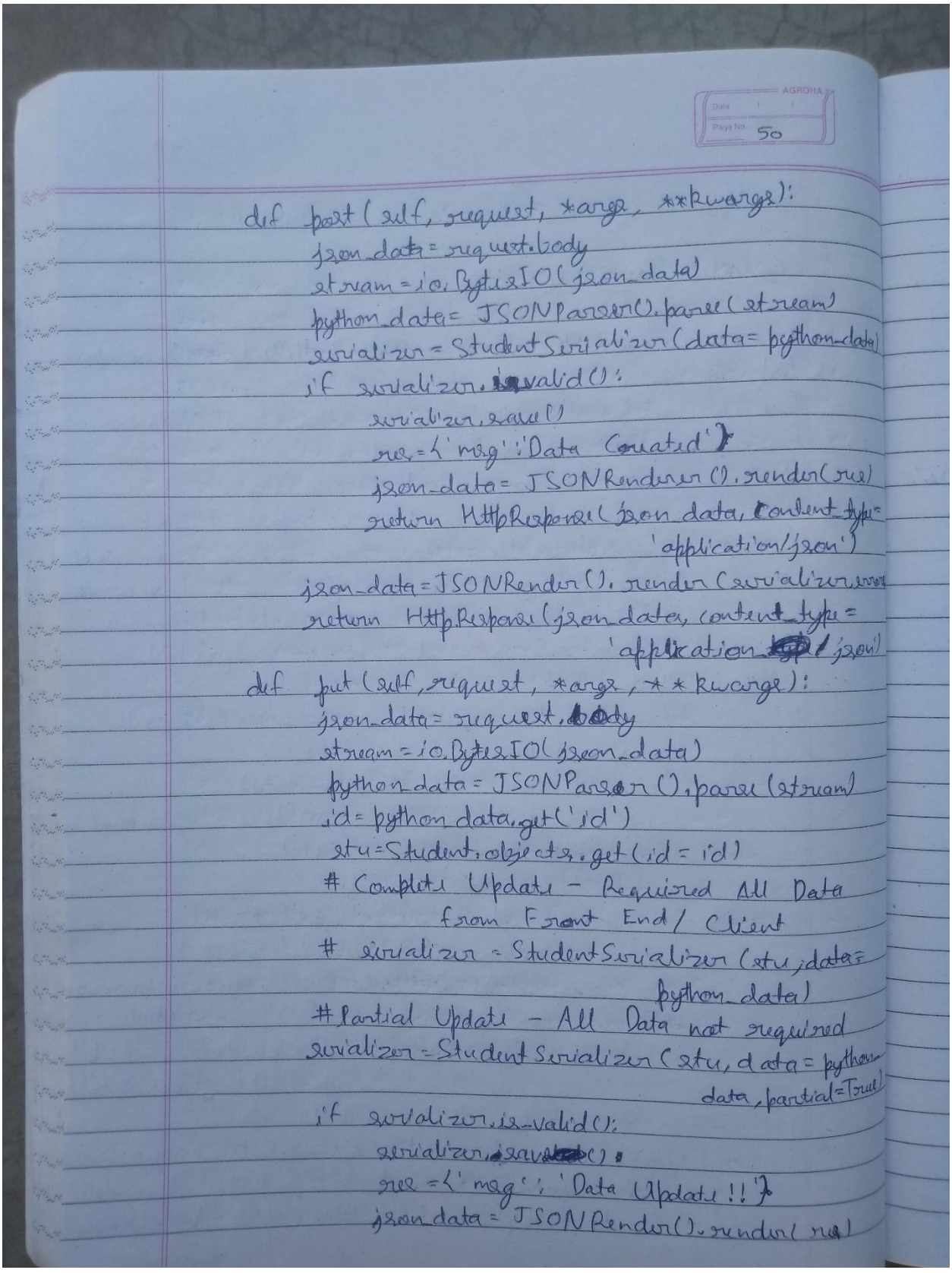
```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns=[
    path('admin/', admin.site.urls),
    path('studentapi/', views.student_api),
]
```



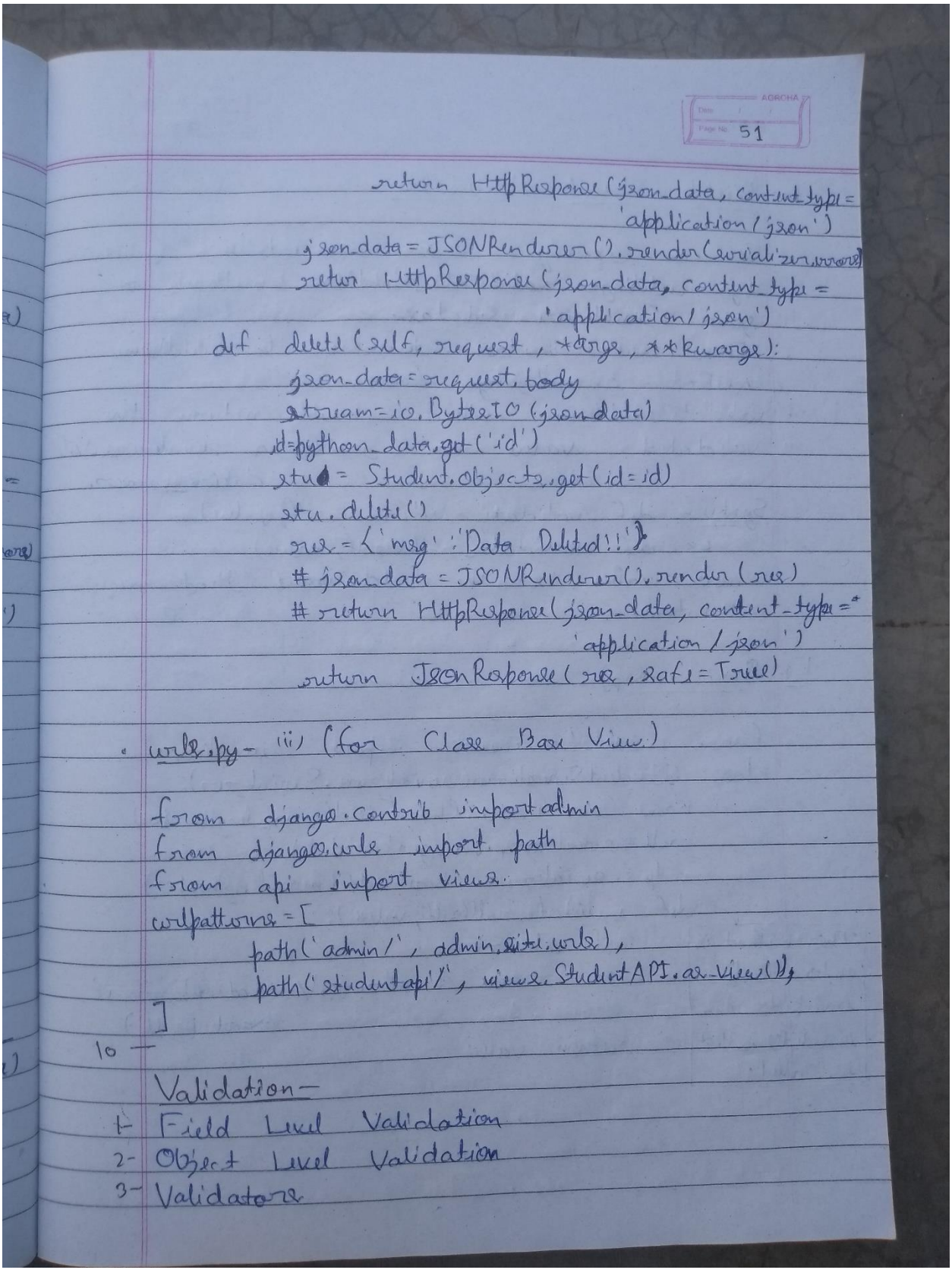
- views.py -

(ii) Class Based View -

```
from django.http.response import HttpResponse, JsonResponse
import io
from rest_framework.parsers import JSONParser
from .models import Student
from .serializers import StudentSerializer
from rest_framework.renderers import JSONRenderer
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator
from django.views import View
@method_decorator(csrf_exempt, name='dispatch')
class StudentAPI(View):
    def get(self, request, *args, **kwargs):
        json_data = request.body
        stream = io.BytesIO(json_data)
        python_data = JSONParser().parse(stream)
        id = python_data.get('id', None)
        if id is not None:
            stu = Student.objects.get(id=id)
            serializer = StudentSerializer(stu)
            json_data = JSONRenderer().render(serializer
                                              .data)
            return HttpResponse(json_data, content_t
                                ype='application/json')
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many=True)
        json_data = JSONRenderer().render(serializer.data)
        return HttpResponse(json_data, content_type='application/
                            json')
```



```
def post(self, request, *args, **kwargs):
    json_data = request.body
    stream = io.BytesIO(json_data)
    python_data = JSONParser().parse(stream)
    serializer = StudentSerializer(data=python_data)
    if serializer.is_valid():
        serializer.save()
        res = {'msg': 'Data Created'}
        json_data = JSONRenderer().render(res)
        return HttpResponse(json_data, content_type=
                            'application/json')
    json_data = JSONRender().render(serializer.errors
    return HttpResponse(json_data, content_type=
                        'application/json')
def put(self, request, *args, **kwargs):
    json_data = request.body
    stream = io.BytesIO(json_data)
    python_data = JSONParser().parse(stream)
    id = python_data.get('id')
    stu = Student.objects.get(id = id)
    # Complete Update - Required All Data
                from Front End / Client
    # serializer = StudentSerializer(stu, data=
                                     python_data)
    # Partial Update - All Data not required
    serializer = StudentSerializer(stu, data = python_
                                   data, partial=True)
    if serializer.is_valid():
        serializer.save()
        res = {'msg': 'Data Updated !!'}
        json_data = JSONRender().render(res)
```



```
                return HttpResponse(json_data, content_type=
                                    'application/json')
        json_data = JSONRenderer().render(serializer.errors)
        return HttpResponse(json_data, content_type =
                            'application/json')
    def delete(self, request, *args, **kwargs):
        json_data = request.body
        stream = io.BytesIO(json_data)
        id = python_data.get('id')
        stu = Student.objects.get(id=id)
        stu.delete()
        res = {'msg': 'Data Deleted!!'}
        # json_data = JSONRenderer().render(res)
        # return HttpResponse(json_data, content_type=
                              'application/json')
        return JsonResponse(res, safe=True)
```

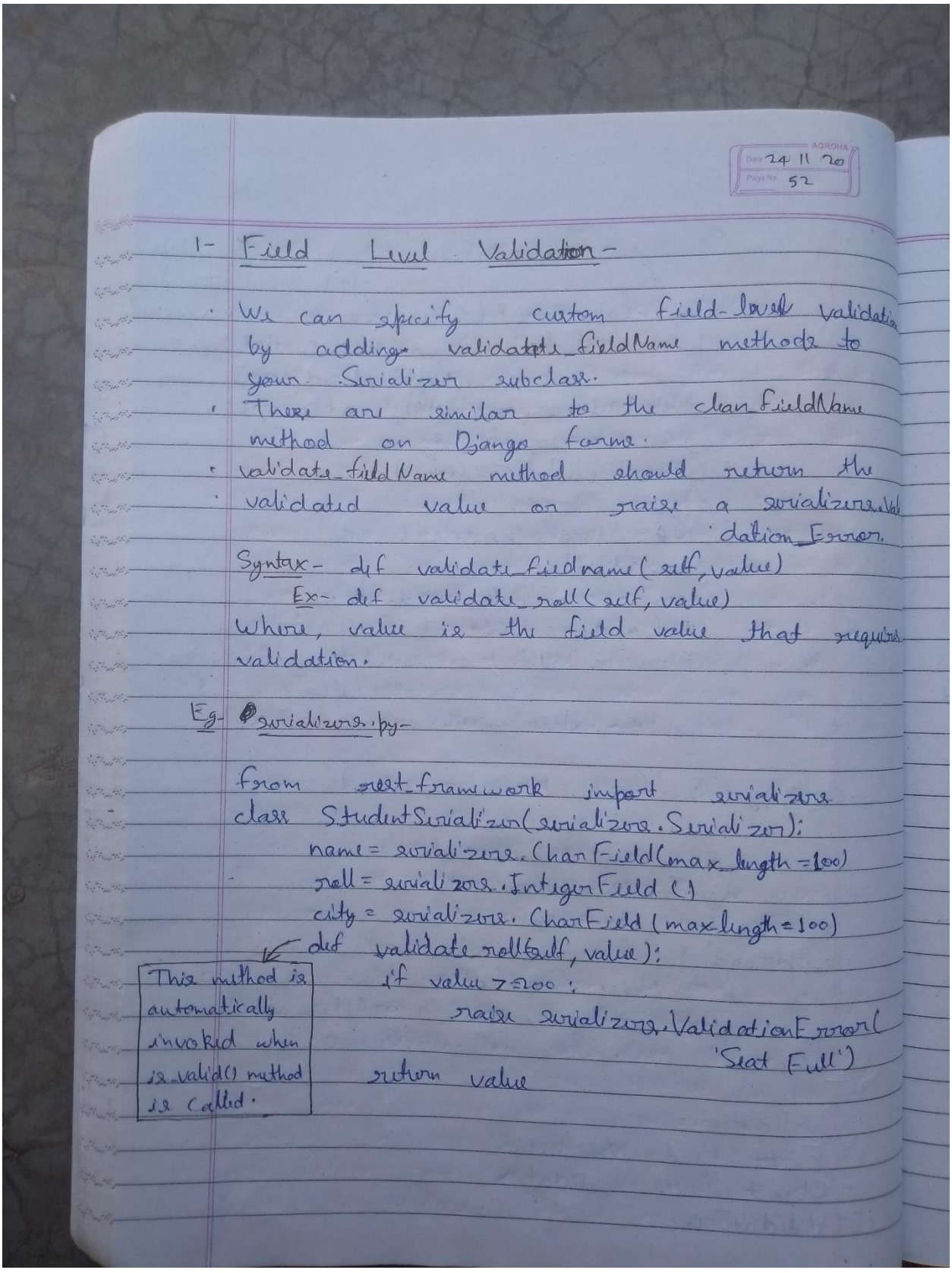
- urls.py - (ii) (for Class Base View)

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
        path('admin/', admin.site.urls),
        path('studentapi/', views.StudentAPI.as_view()),
]
```

10 -

Validation -

1- Field Level Validation
2- Object Level Validation
3- Validators



1- Field Level Validation -

- We can specify custom field-level validation by adding validate_fieldName methods to your Serializer subclass.
- These are similar to the clean_fieldName method on Django forms.
- validate_fieldName method should return the validated value or raise a serializers.ValidationError.

Syntax - def validate_fieldname(self, value)

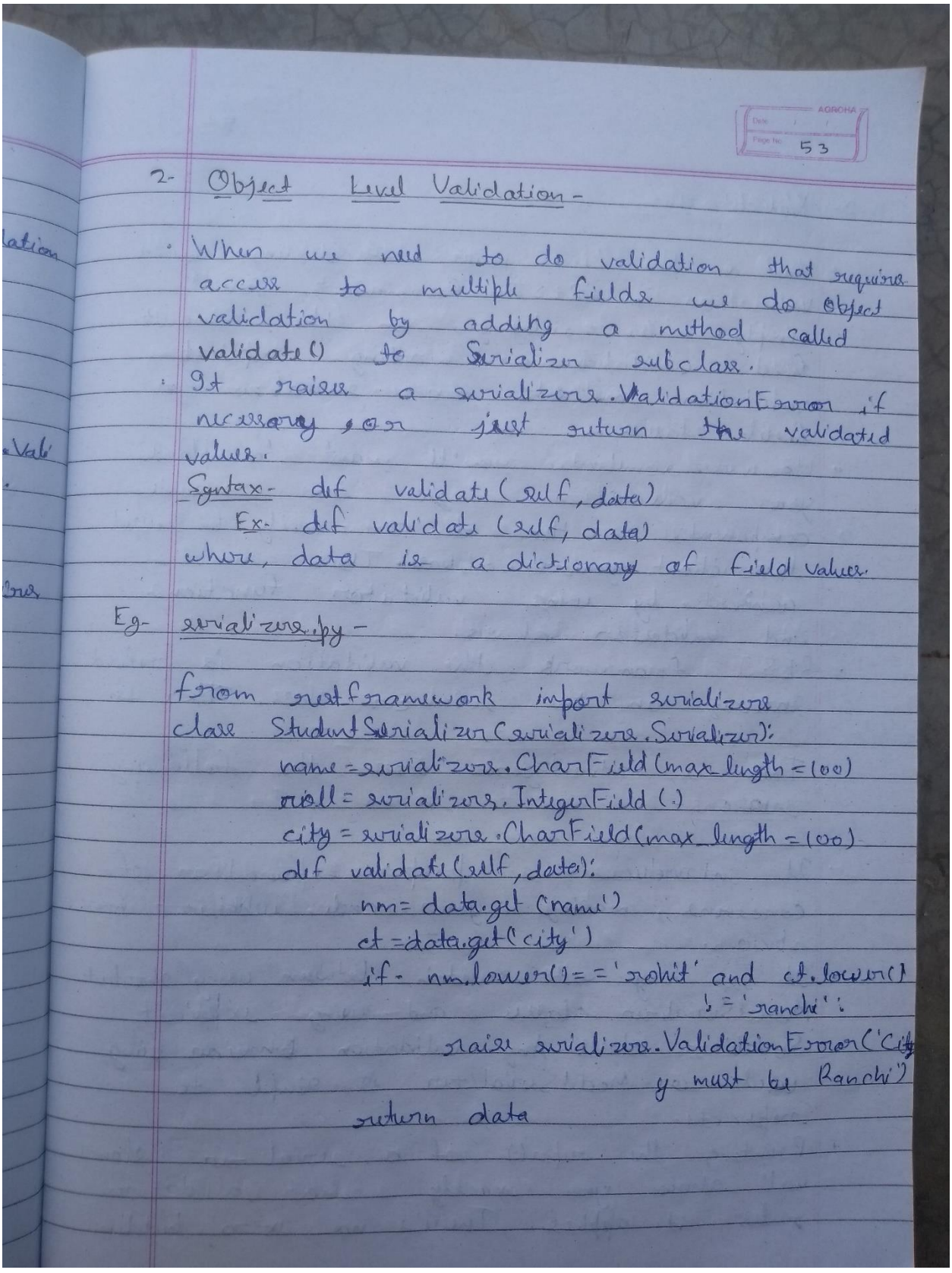
Ex - def validate_roll(self, value)

Where, value is the field value that requires validation.

Eg - serializers.py -

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
    def validate_roll(self, value):
        if value >= 200:
            raise serializers.ValidationError('Seat Full')
        return value
```

This method is automatically invoked when is_valid() method is called.



2- Object Level Validation -

- When we need to do validation that requires access to multiple fields we do Object validation by adding a method called validate() to Serializer subclass.
- It raise a serializers.ValidationError if necessary, or just return the validated values.

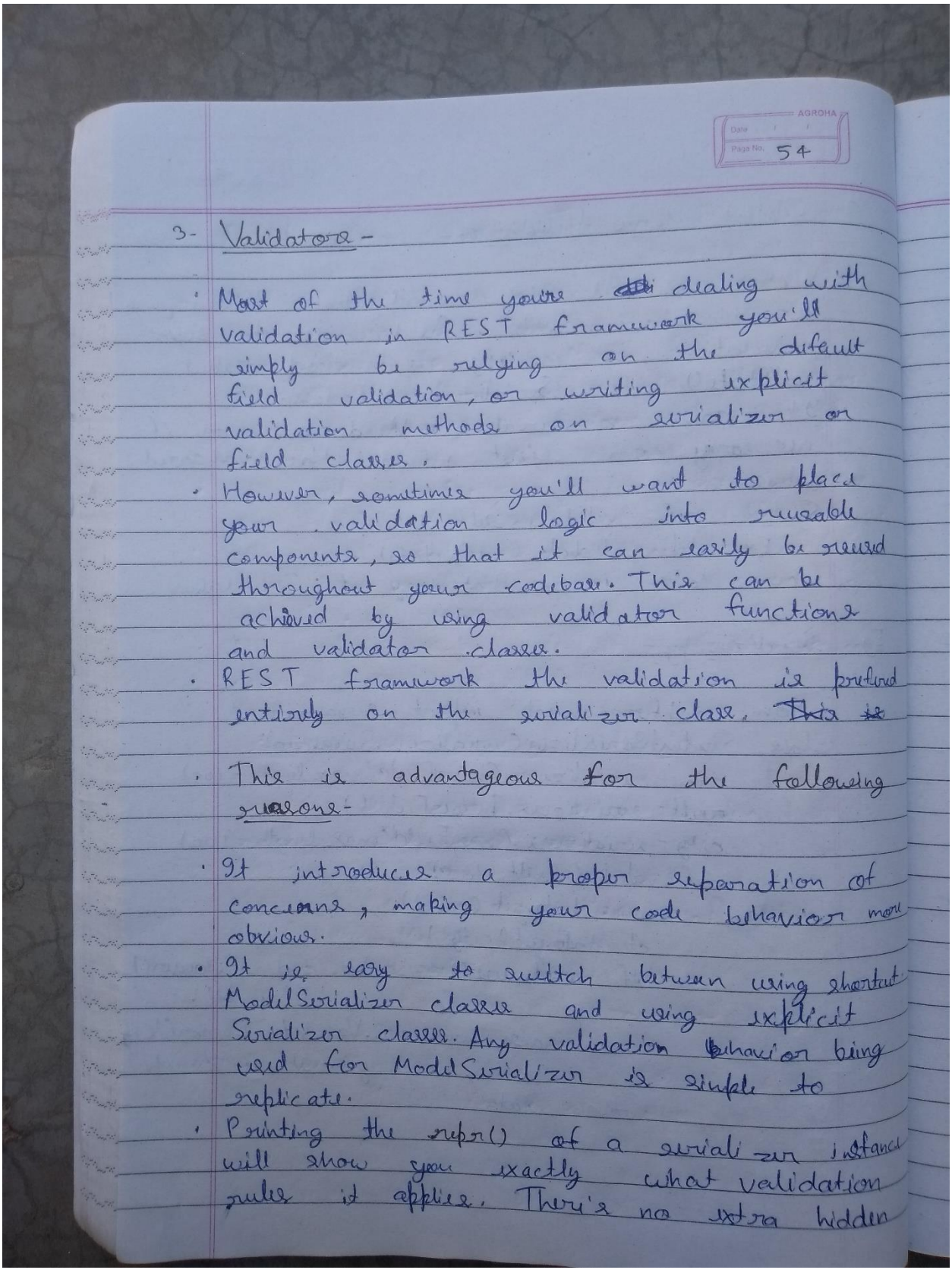
Syntax- def validate(self, data)

Ex. def validate(self, data)

where, data is a dictionary of field values.

Eg- serializers.py -

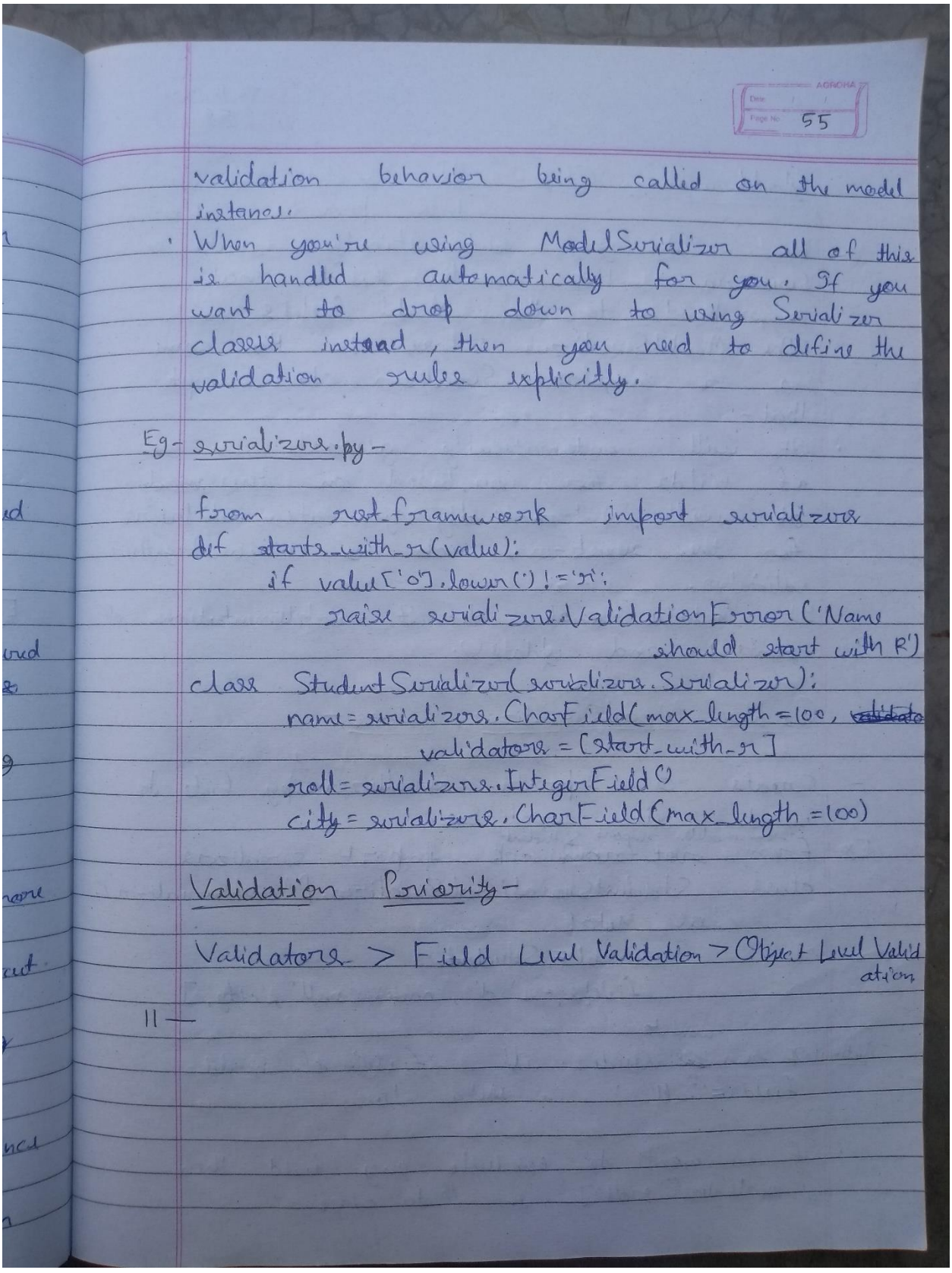
```
from rest framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
    def validate(self, data):
        nm = data.get('name')
        ct = data.get('city')
        if nm.lower() == 'rohit' and ct.lower()
                                    != 'ranchi':
            raise serializers.ValidationError('City
                                    y must be Ranchi')
        return data
```



AGROHA
Date / /
Page No. 54

3- Validators -

- Most of the time you're dealing with validation in REST framework you'll simply be relying on the default field validation, or writing explicit validation methods on serializer or field classes.
- However, sometimes you'll want to place your validation logic into reusable components, so that it can easily be reused throughout your codebase. This can be achieved by using validator functions and validator classes.
- REST framework the validation is pushed entirely on the serializer class.
- This is advantageous for the following reasons -
- It introduces a proper separation of concerns, making your code behavior more obvious.
- It is easy to switch between using shortcut ModelSerializer classes and using explicit Serializer classes. Any validation behavior being used for ModelSerializer is simple to replicate.
- Printing the repr() of a serializer instance will show you exactly what validation rules it applies. There's no extra hidden



validation behavior being called on the model instance.

- When you're using ModelSerializer all of this is handled automatically for you. If you want to drop down to using Serializer classes instead, then you need to define the validation rules explicitly.

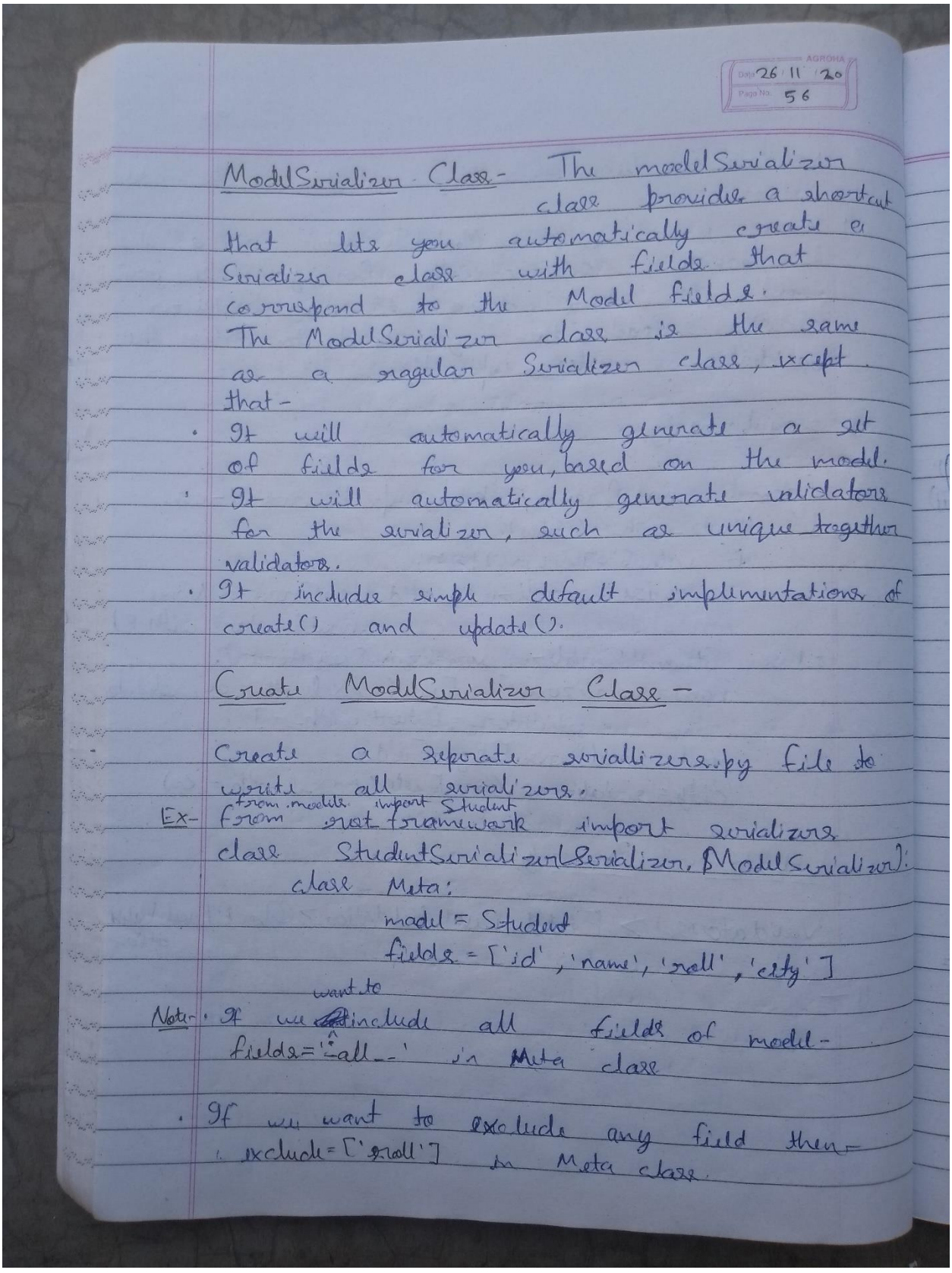
Eg- serializers.py -

```
from rest_framework import serializers
def starts_with_r(value):
    if value[0].lower() != 'r':
        raise serializers.ValidationError('Name should start with R')
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100, validators=[starts_with_r])
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

Validation Poriority -

Validators > Field Level Validation > Object Level Validation

11 —



ModelSerializer Class - The modelSerializer class provides a shortcut that lets you automatically create a Serializer class with fields that correspond to the Model fields.
The ModelSerializer class is the same as a regular Serializer class, except that -

- It will automatically generate a set of fields for you, based on the model.
- It will automatically generate validators for the serializer, such as unique_together validators.
- It includes simple default implementations of create() and update().

Create ModelSerializer Class -

Create a separate serializers.py file to write all serializers.

Ex- from .models import Student
from rest_framework import serializers
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll', 'city']

Note - If we want to include all fields of model - fields = '__all__' in Meta class

- If we want to exclude any field then - exclude = ['roll'] in Meta class.

Note- We cannot set both 'fields' and 'exclude' options on serializer at the same time we use any of one at a time.

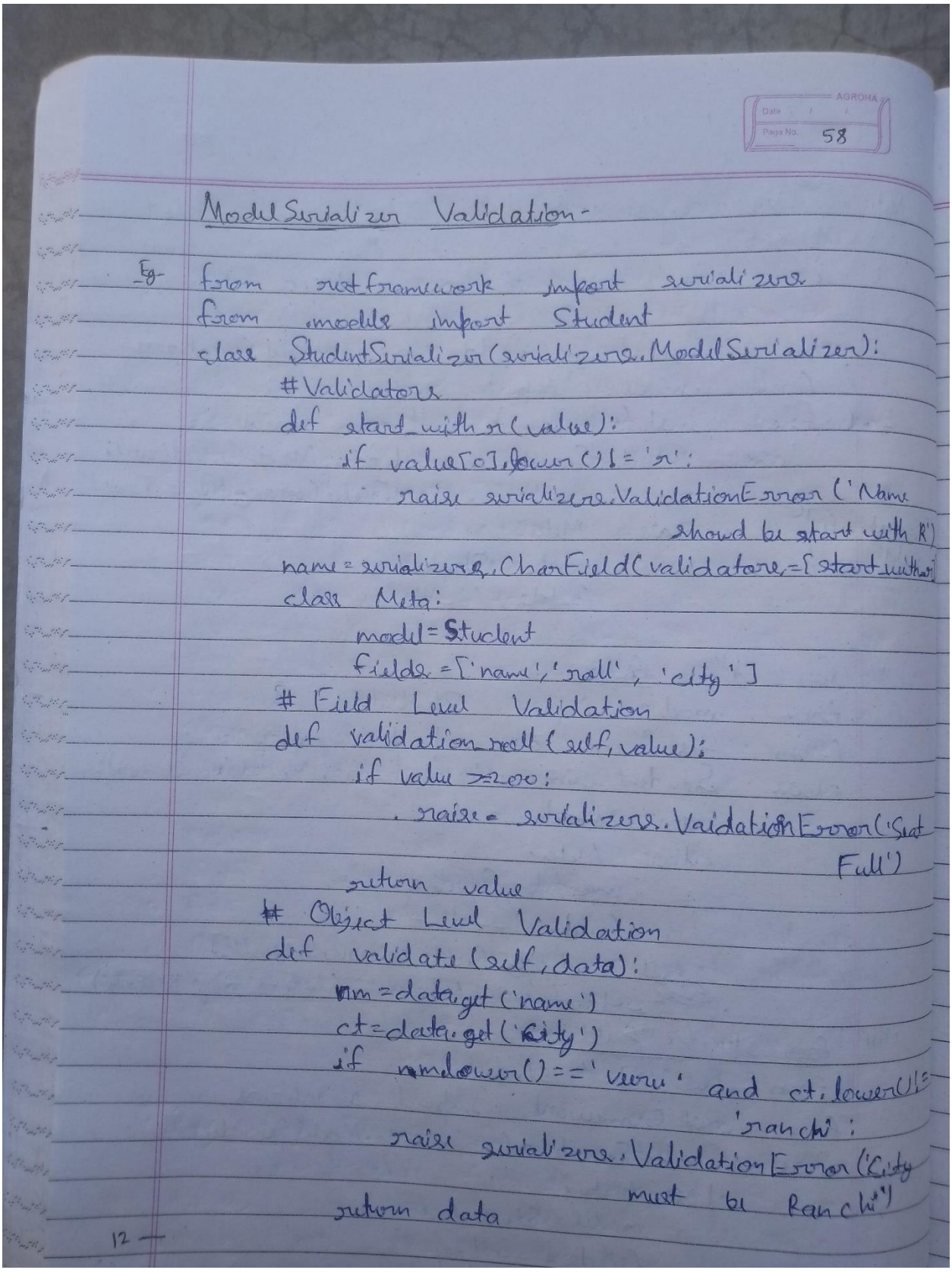
Set some arguments in field of ModelSerializer -

Ex-(i) from .models import Student
from rest_framework import serializers
class StudentSerializer(serializers.ModelSerializer):
name = serializers.CharField(read_only = True)
class Meta:
model = Student
fields = '__all__'

For specific field

(ii) for all fields -
from .models import Student
from rest_framework import serializers
class StudentSerializer(serializers.ModelSerializer):
~~name = serializer~~
class Meta:
model = Student
fields = '__all__'
read_only_fields = ['name', 'roll']
~~extra_fields = {'name': {'read_only': True}}~~

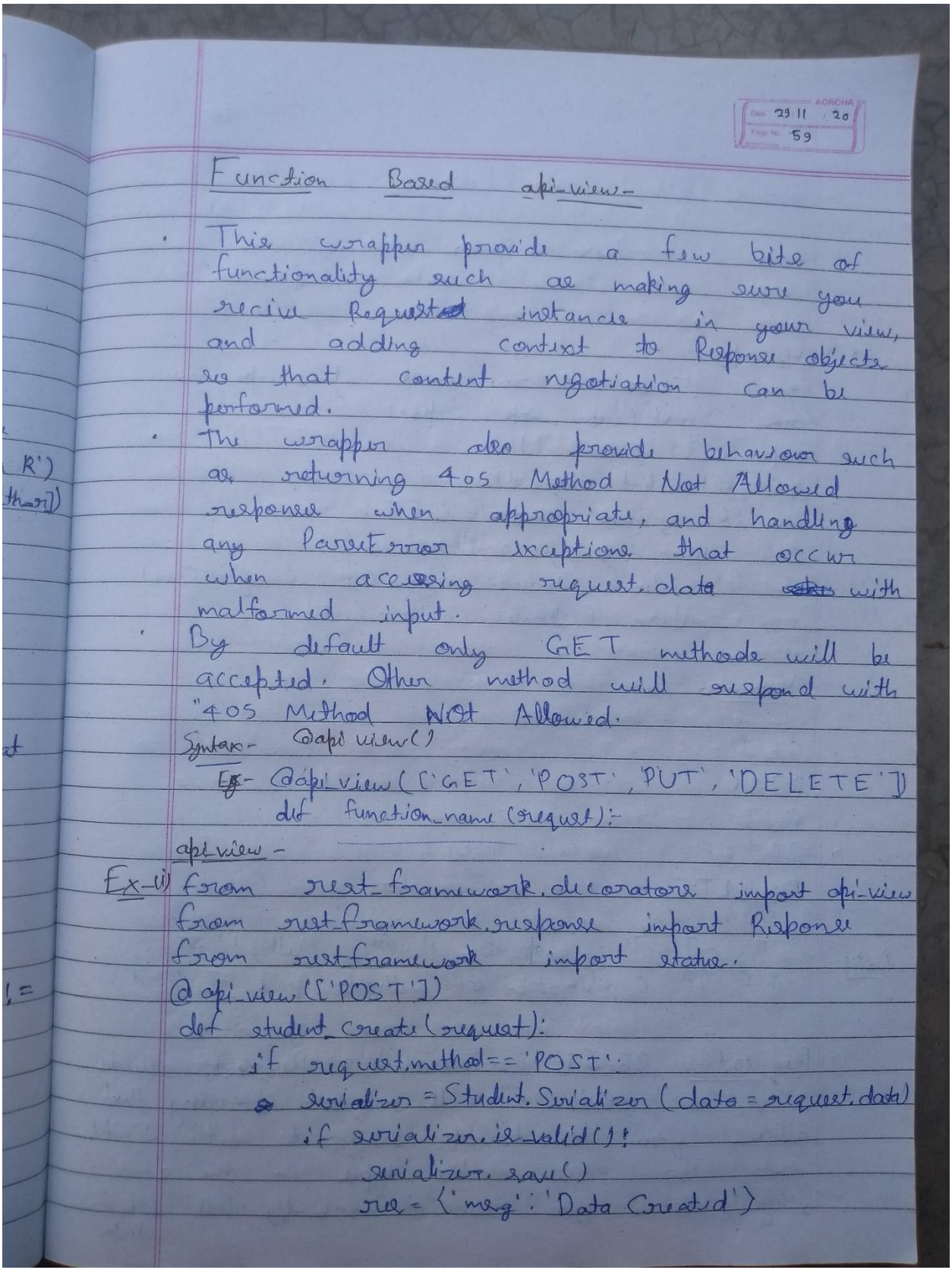
(iii) for specific fields -
from .model import student
from rest_framework import serializers
class StudentSerializer(serializers.ModelSerializer):
class Meta:
model = Student
fields = '__all__'
extra_kwargs = {'name': {'read_only': True}}



ModelSerializer Validation-

Eg-

```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.ModelSerializer):
    #Validators
    def start_with_r(value):
        if value[0].lower() != 'r':
            raise serializers.ValidationError('Name
                                    should be start with R')
    name = serializers.CharField(validators=[start_with_r])
    class Meta:
        model = Student
        fields = ['name', 'roll', 'city']
    # Field Level Validation
    def validate_roll(self, value):
        if value >= 200:
            raise serializers.ValidationError('Seat
                                    Full')
        return value
    # Object Level Validation
    def validate(self, data):
        nm = data.get('name')
        ct = data.get('city')
        if nm.lower() == 'virat' and ct.lower() !=
                                    'ranchi':
            raise serializers.ValidationError('City
                                    must be Ranchi')
        return data
```



Function Based api_view -

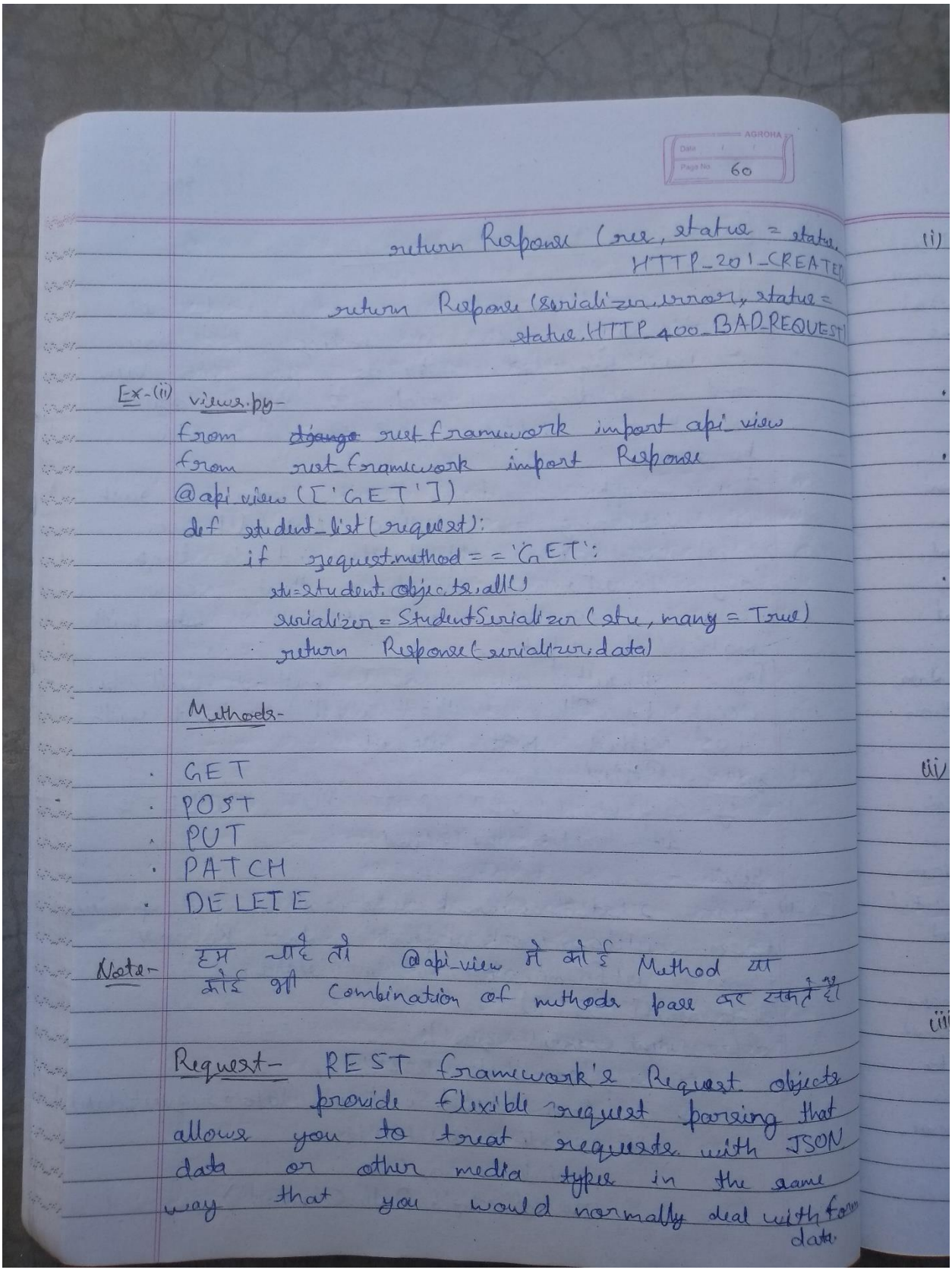
- This wrapper provide a few bits of functionality such as making sure you recive Request instance in your view, and adding context to Response objects so that content negotiation can be performed.
- The wrapper also provide behaviour such as returning 405 Method Not Allowed response when appropriate, and handling any ParseError exceptions that occur when accessing request.data with malformed input.
- By default only GET methods will be accepted. Other method will respond with "405 Method NOt Allowed.

Syntax- @api_view()

Eg- @api_view(['GET', 'POST', 'PUT', 'DELETE'])
def function_name(request):

api_view -

Ex-i)
```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status.
@api_view(['POST'])
def student_create(request):
    if request.method=='POST':
        serializer = Student.Serializer(data = request.data)
        if serializer.is_valid():
            serializer.save()
            res = {'msg': 'Data Created'}
```



return Response (ser, status = status.
HTTP_201_CREATED)
return Response (serializer.errors, status =
status.HTTP_400_BAD_REQUEST)

Ex-(ii) views.py-

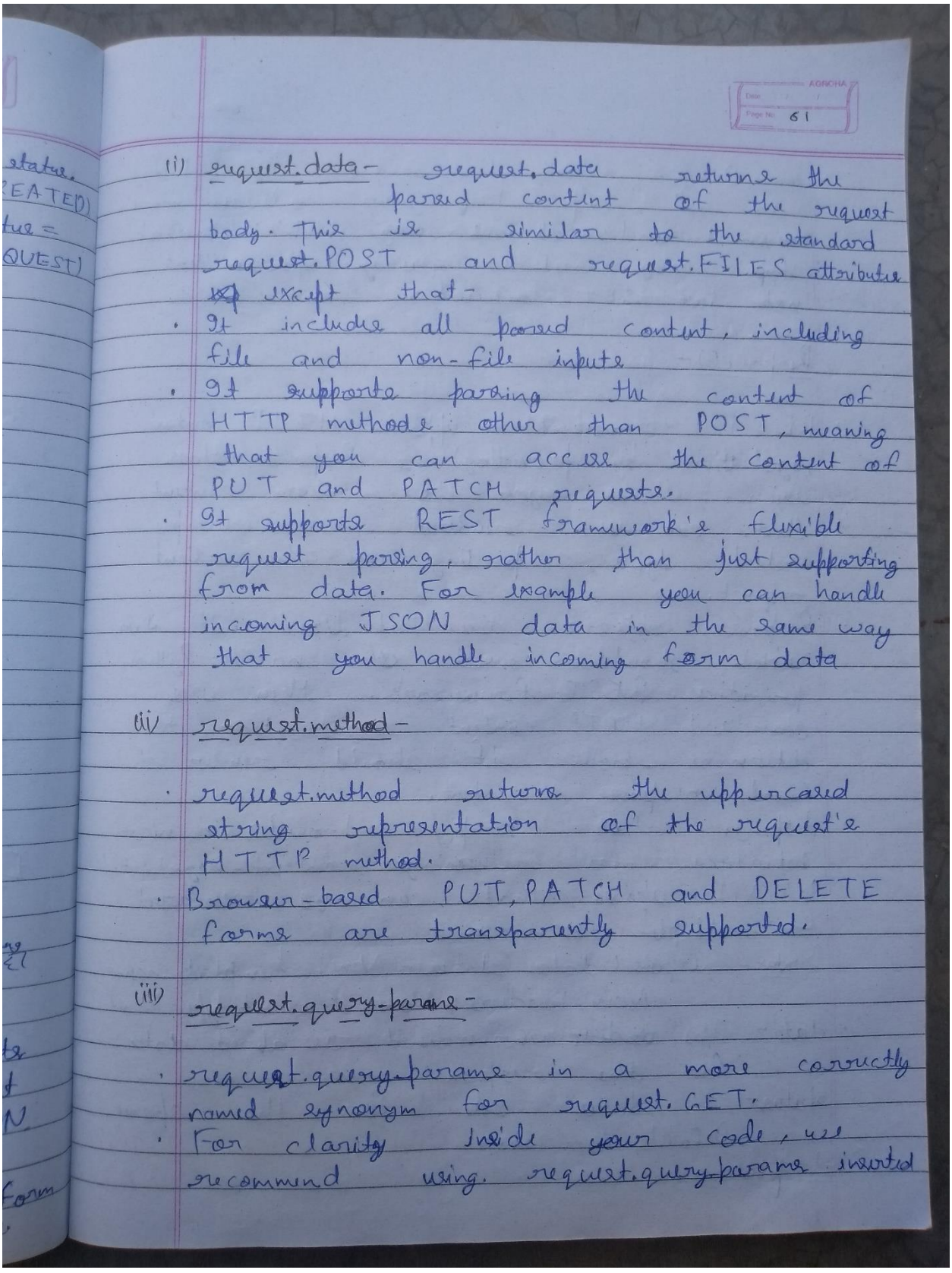
```
from django.rest_framework import api_view
from rest_framework import Response
@api_view(['GET'])
def student_list(request):
    if request.method == 'GET':
        stu=Student.objects.all()
        serializer = StudentSerializer(stu, many = True)
        return Response(serializer.data)
```

Methods-

- GET
- POST
- PUT
- PATCH
- DELETE

Note- हम चाहे तो @api_view में कोई Method या कोई भी Combination of methods pass कर सकते हैं

Request- REST framework's Request objects provide flexible request parsing that allows you to treat requests with JSON data or other media types in the same way that you would normally deal with form data.



(i) request.data - request.data returns the parsed content of the request body. This is similar to the standard request.POST and request.FILES attributes except that -

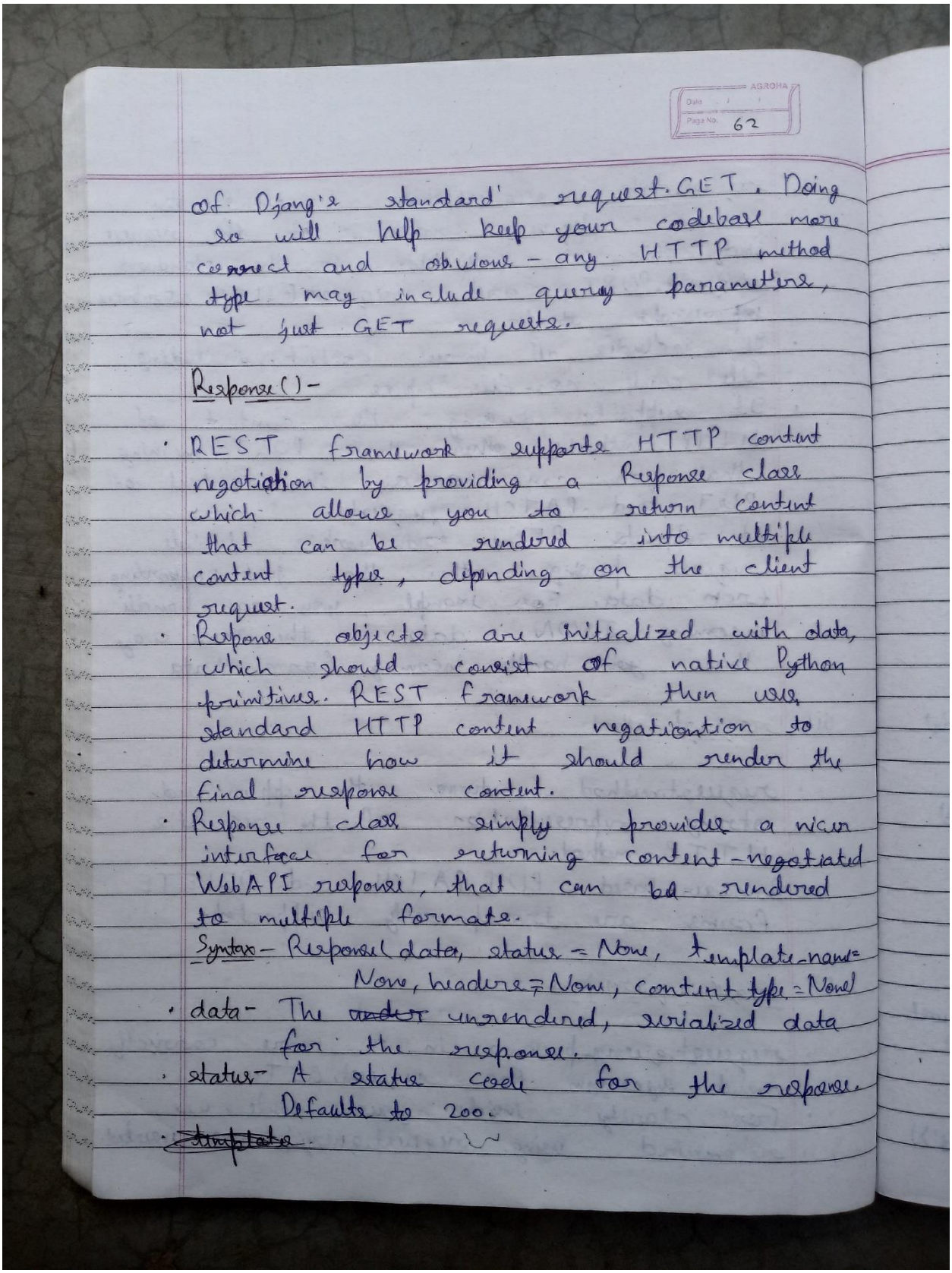
- It includes all parsed content, including file and non-file inputs
- It supports parsing the content of HTTP methods other than POST, meaning that you can access the content of PUT and PATCH requests.
- It supports REST framework's flexible request parsing, rather than just supporting form data. For example you can handle incoming JSON data in the same way that you handle incoming form data

(ii) request.method -

- request.method returns the uppercased string representation of the request's HTTP method.
- Browser-based PUT, PATCH and DELETE forms are transparently supported.

(iii) request.query_params -

- request.query_params is a more correctly named synonym for request.GET.
- For clarity inside your code, we recommend using request.query_params instead



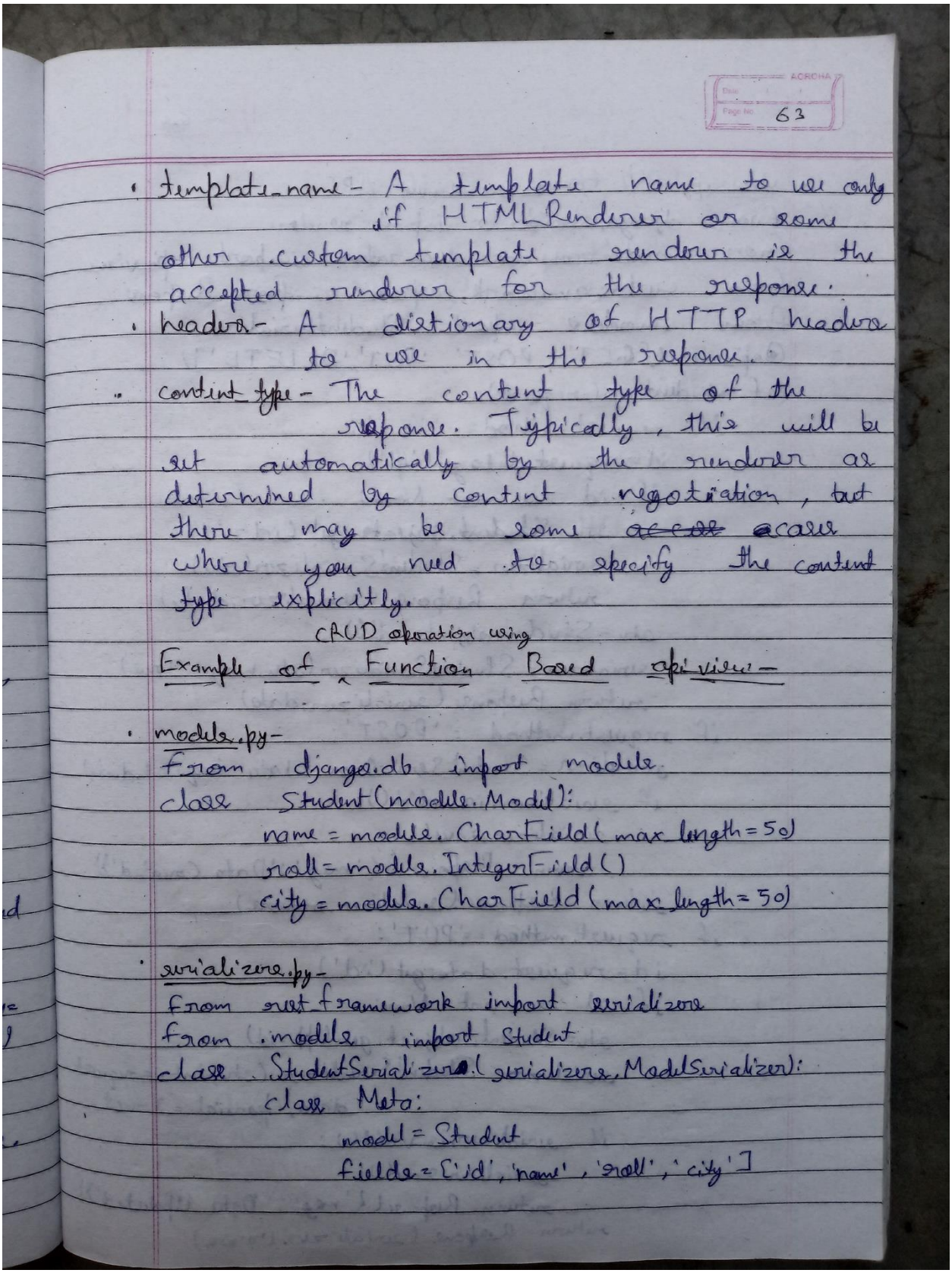
of Djang's standard' request.GET. Doing so will help keep your codebase more correct and obvious - any HTTP method type may include query parameters, not just GET requests.

Response()-

- REST framework supports HTTP content negotiation by providing a Response class which allows you to return content that can be rendered into multiple content types, depending on the client request.
- Response objects are initialized with data, which should consist of native Python primitives. REST framework then uses standard HTTP content negotiation to determine how it should render the final response content.
- Response class simply provides a nicer interface for returning content-negotiated Web API responses, that can be rendered to multiple formats.

Syntax- Response(data, status=None, template_name=None, headers=None, content_type=None)

- data- The unrendered, serialized data for the response.
- status- A status code for the response. Defaults to 200.
- ~~template~~



- template_name - A template name to use only if HTMLRenderer or some other custom template renderer is the accepted renderer for the response.
- headers - A dictionary of HTTP headers to use in the response.
- content_type - The content type of the response. Typically, this will be set automatically by the renderer as determined by content negotiation, but there may be some ~~a~~ cases where you need to specify the content type explicitly.

CRUD operation using

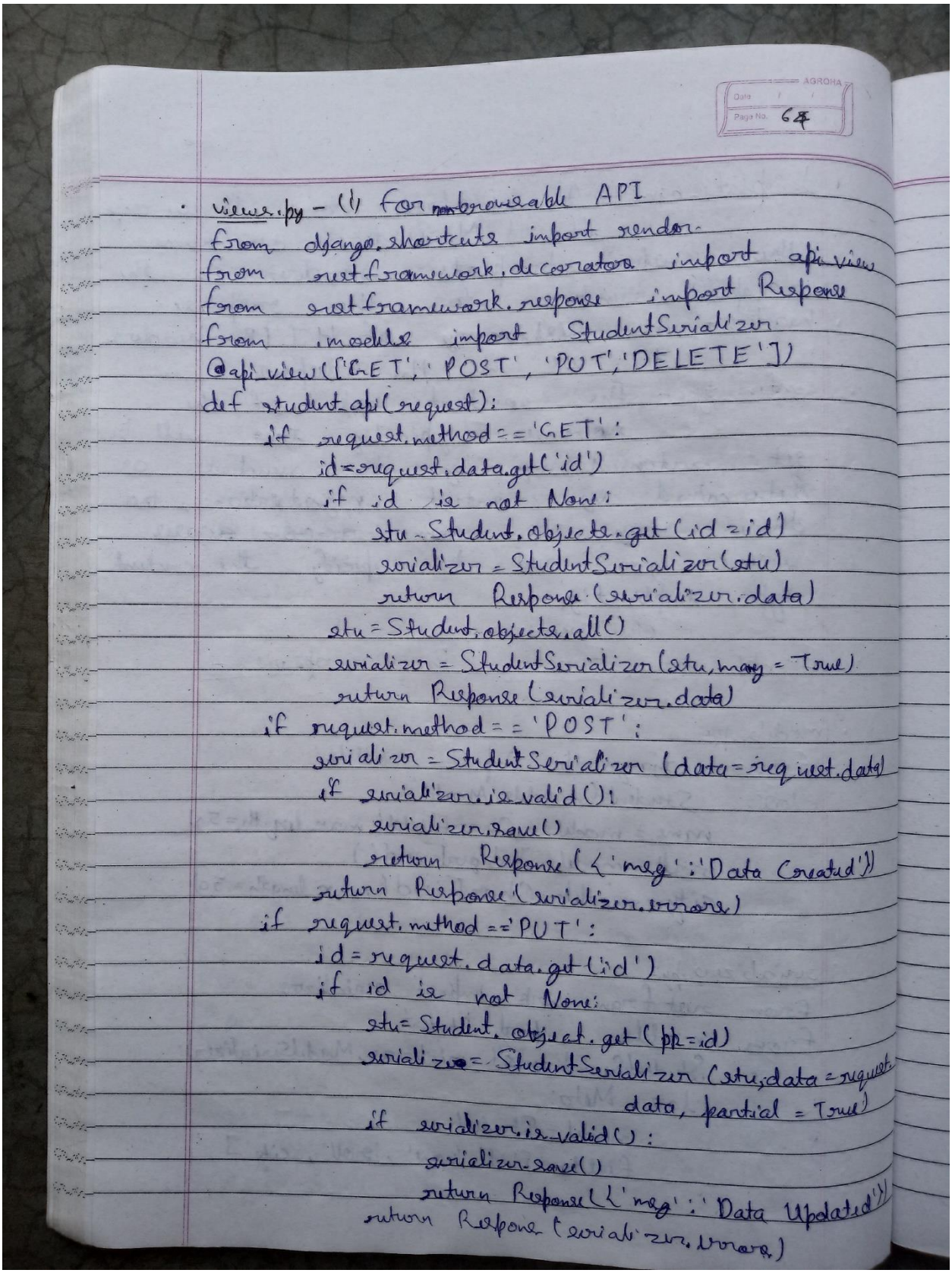
Example of Function Based api view -

- models.py -

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.IntegerField()
    city = models.CharField(max_length=50)
```

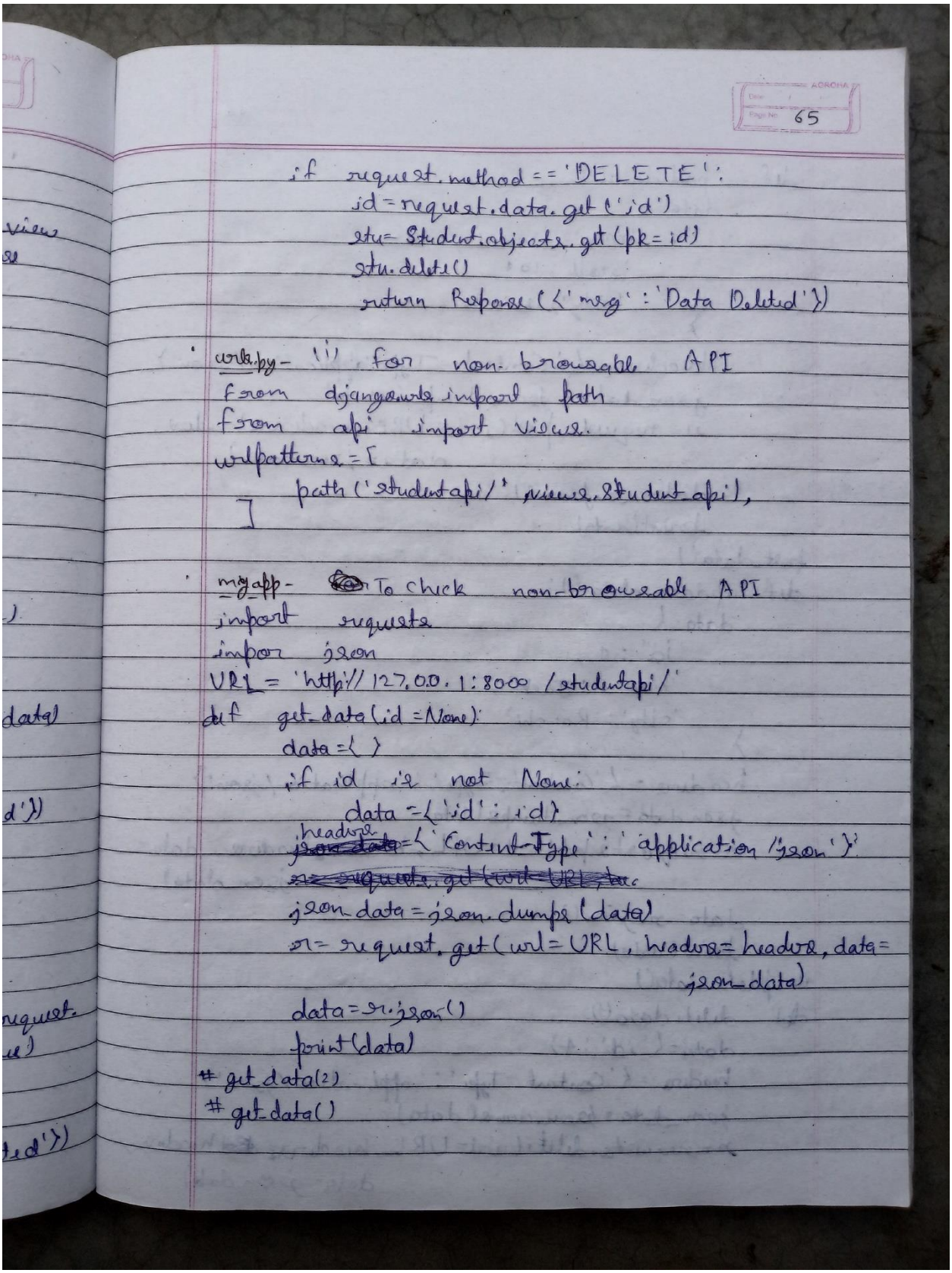
- serializers.py -

```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll', 'city']
```



views.py - (1) for non browsable API

```
from django.shortcuts import render
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import StudentSerializer
@api_view(['GET', 'POST', 'PUT', 'DELETE'])
def student_api(request):
    if request.method == 'GET':
        id = request.data.get('id')
        if id is not None:
            stu = Student.objects.get(id = id)
            serializer = StudentSerializer(stu)
            return Response(serializer.data)
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many = True)
        return Response(serializer.data)
    if request.method == 'POST':
        serializer = StudentSerializer(data = request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg': 'Data Created'})
        return Response(serializer.errors)
    if request.method == 'PUT':
        id = request.data.get('id')
        if id is not None:
            stu = Student.object.get(pk = id)
            serializer = StudentSerializer(stu, data = request.
                                           data, partial = True)
            if serializer.is_valid():
                serializer.save()
                return Response({'msg': 'Data Updated'})
        return Response(serializer.errors)
```



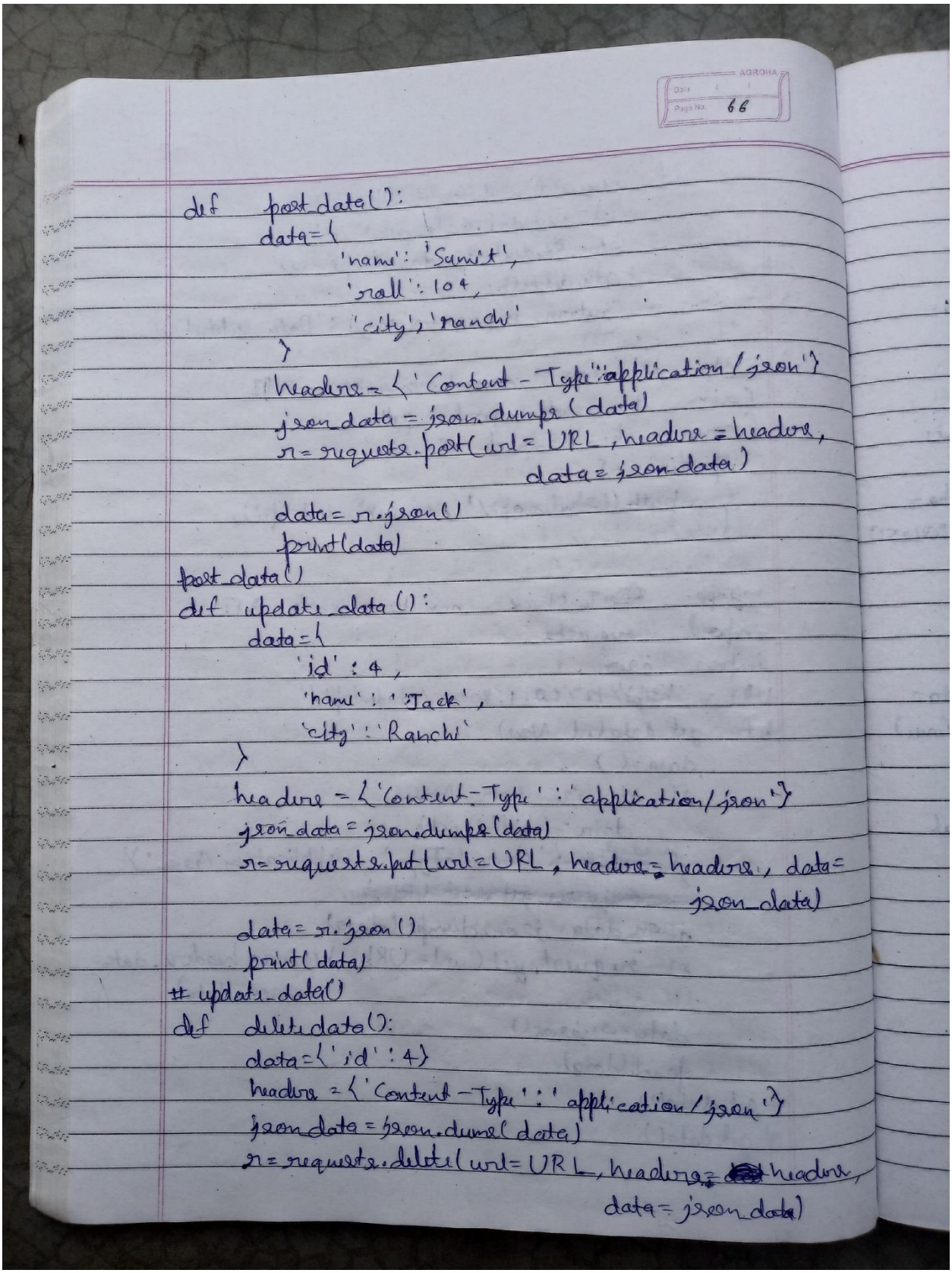
```
if request.method == 'DELETE':
    id = request.data.get('id')
    stu = Student.objects.get(pk=id)
    stu.delete()
    return Response({'msg': 'Data Deleted'})
```

urls.py - (i) for non-browsable API

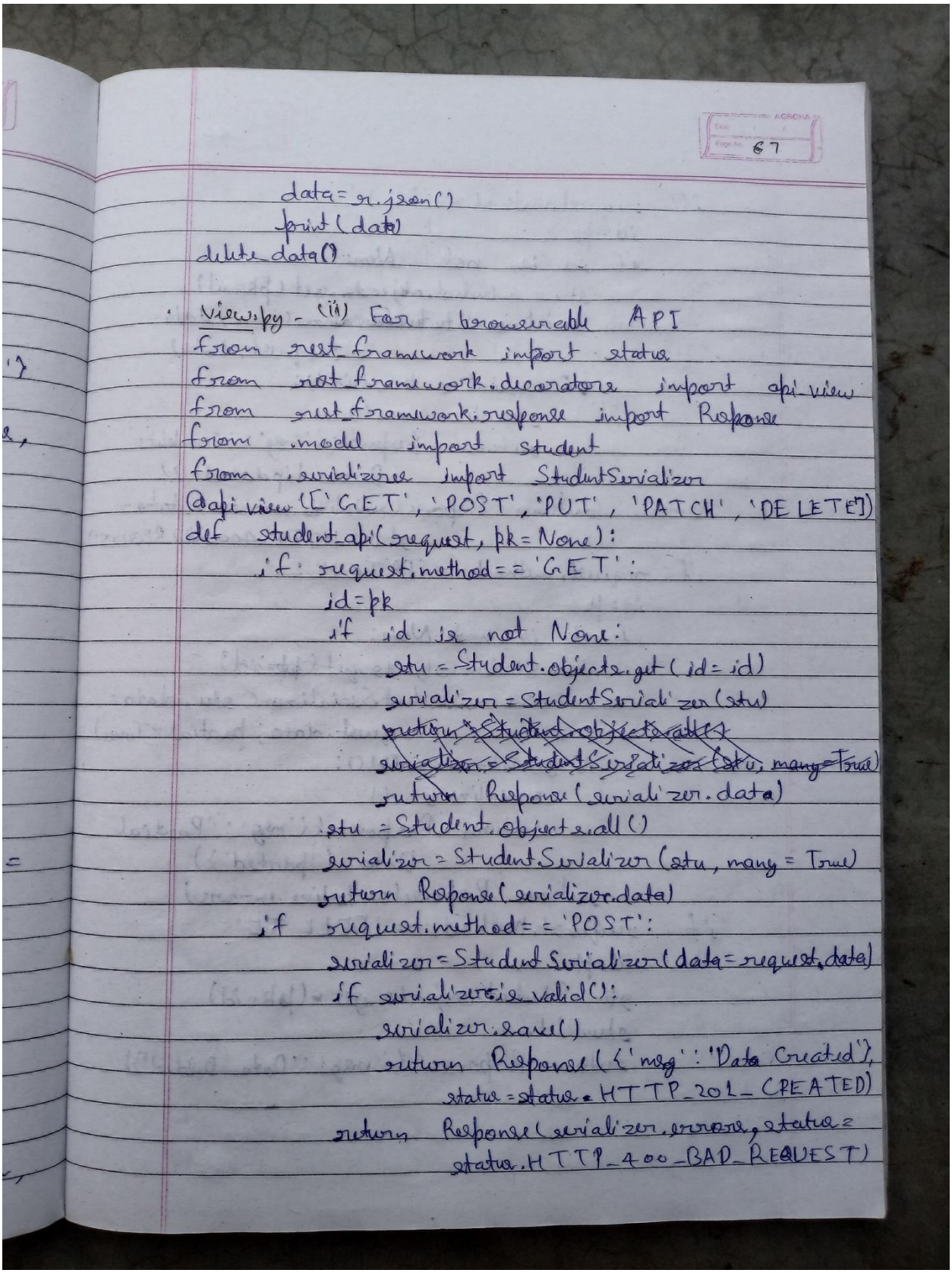
```
from django.urls import path
from api import views
urlpatterns = [
    path('studentapi/', views.Student_api),
]
```

myapp - To check non-browsable API

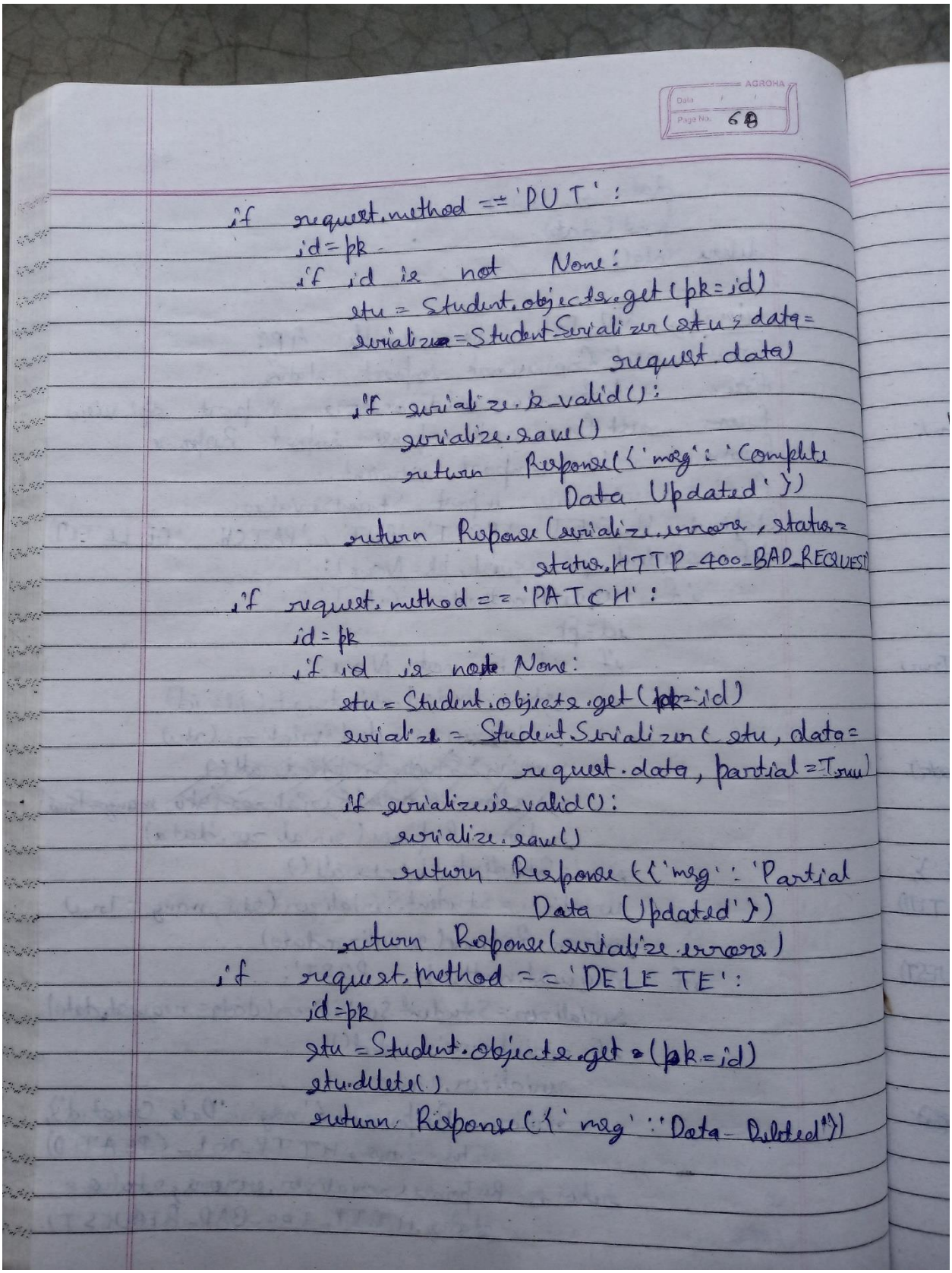
```
import requests
import json
URL = 'http://127.0.0.1:8000/studentapi/'
def get_data(id=None):
    data = {}
    if id is not None:
        data = {'id': id}
    headers = {'Content-Type': 'application/json'}
    json_data = json.dumps(data)
    r = requests.get(url=URL, headers=headers, data=
                     json_data)
    data = r.json()
    print(data)
# get_data(2)
# get_data()
```



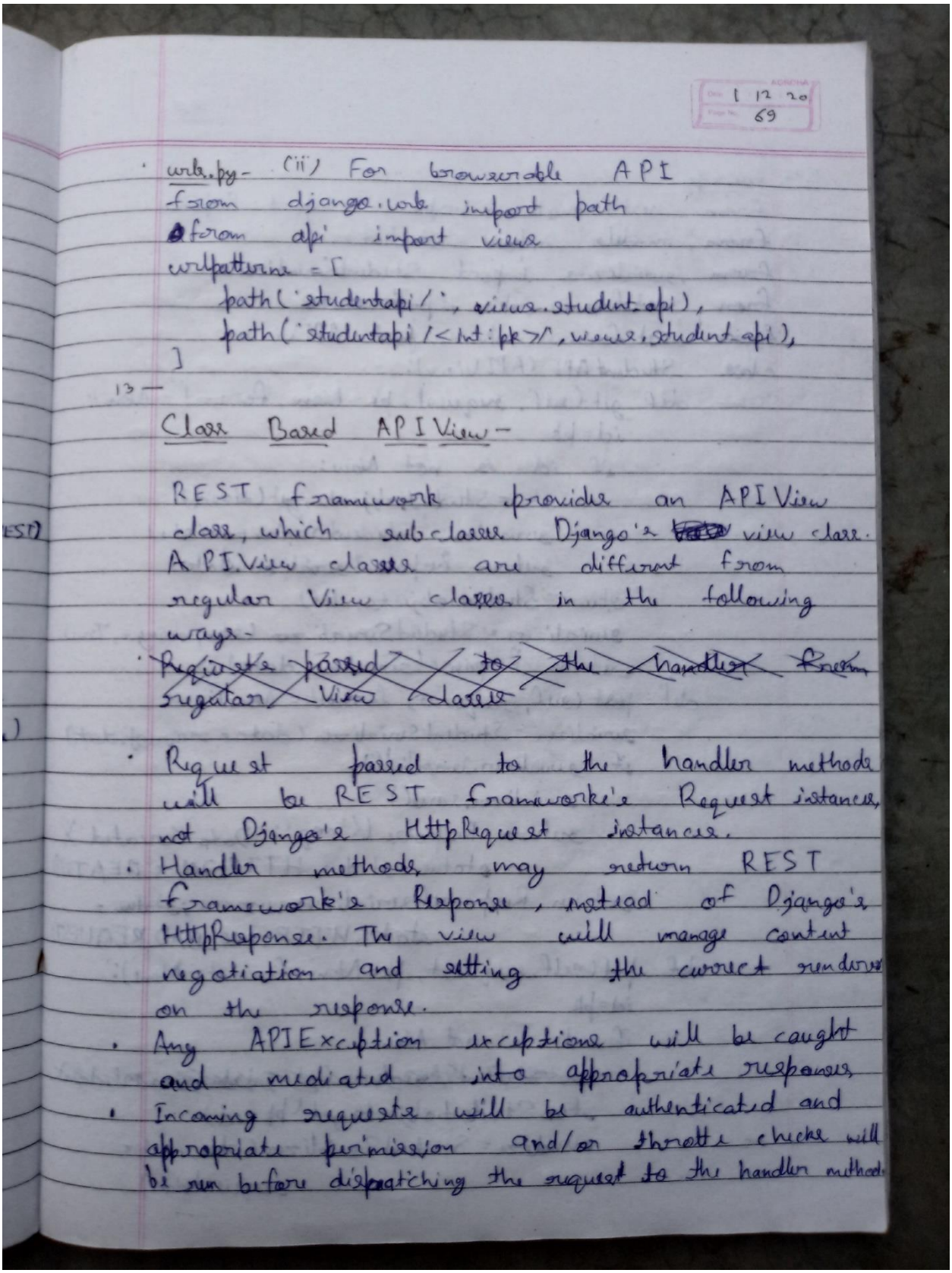
```
def post_data():
    data={
        'name': 'Sumit',
        'roll': 104,
        'city': 'ranchi'
    }
    headers={'Content-Type':'application/json'}
    json_data = json.dumps(data)
    r=requests.post(url=URL, headers=headers,
                    data=json_data)

    data=r.json()
    print(data)
post_data()
def update_data():
    data={
        'id': 4,
        'name': 'Jack',
        'city': 'Ranchi'
    }
    headers={'Content-Type': 'application/json'}
    json_data=json.dumps(data)
    r=requests.put(url=URL, headers=headers, data=
                   json_data)
    data=r.json()
    print(data)
# update_data()
def delete_data():
    data={'id':4}
    headers={'Content-Type': 'application/json'}
    json_data=json.dumps(data)
    r=requests.delete(url=URL, headers=headers,
                      data=json_data)
```



```
        data = r.json()
        print(data)
    delete_data()

• views.py - (ii) For browserable API
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .model import student
from .serializers import StudentSerializer
@api_view(['GET', 'POST', 'PUT', 'PATCH', 'DELETE'])
def student_api(request, pk=None):
    if request.method == 'GET':
        id = pk
        if id is not None:
            stu = Student.objects.get(id=id)
            serializer = StudentSerializer(stu)
            return Response(serializer.data)
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many=True)
        return Response(serializer.data)
    if request.method == 'POST':
        serializer = StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg': 'Data Created'},
                status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=
            status.HTTP_400_BAD_REQUEST)
```



```
if request.method == 'PUT':
    id = pk
    if id is not None:
        stu = Student.objects.get(pk=id)
        serializer = StudentSerializer(stu, data=
                                request.data)
        if serialize.is_valid():
            serialize.save()
            return Response({'msg': 'Complete
                                Data Updated'})
        return Response(serialize.errors, status=
                        status.HTTP_400_BAD_REQUEST)
if request.method == 'PATCH':
    id = pk
    if id is not None:
        stu = Student.objects.get(pk=id)
        serial = StudentSerializer(stu, data=
                        request.data, partial=True)
        if serialize.is_valid():
            serialize.save()
            return Response({'msg': 'Partial
                                Data Updated'})
        return Response(serialize.errors)
if request.method == 'DELETE':
    id = pk
    stu = Student.objects.get(pk=id)
    stu.delete()
    return Response({'msg': 'Data Deleted'})
```



· urls.py - (ii) For browsable API

```
from django.urls import path
from api import views
urlpatterns = [
    path('studentapi/', views.student_api),
    path('studentapi/<int:pk>/', views.student_api),
]
```

13 —

Class Based API View -

REST framework provide an APIView class, which subclasses Django's ~~View~~ view class. APIView classes are different from regular View classes in the following ways -

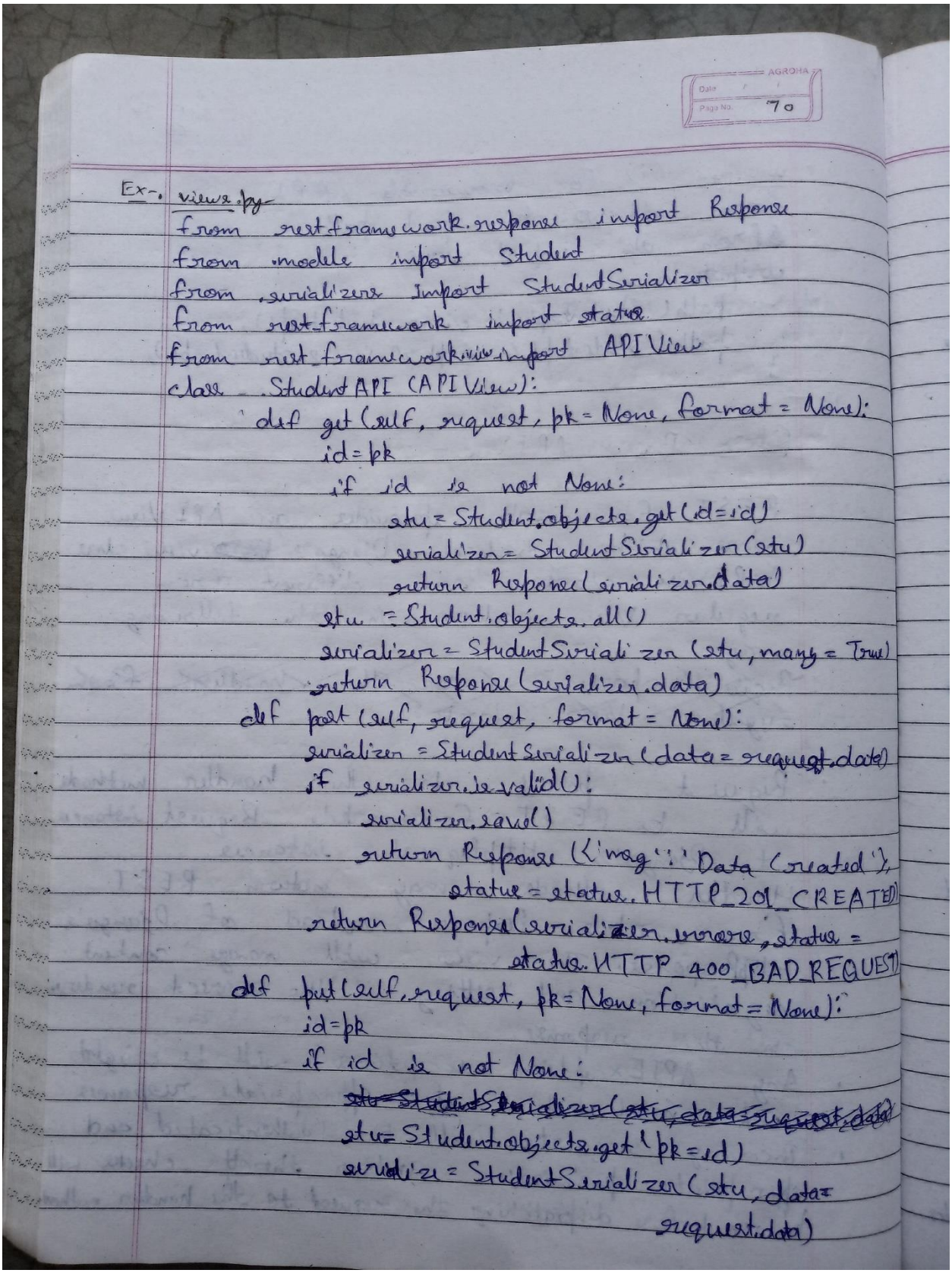
· ~~Requests passed to the handler from regular View class~~

· Request passed to the handler methods will be REST framework's Request instances, not Django's HttpRequest instances.

· Handler methods may return REST framework's Response, instead of Django's HttpResponse. The view will manage content negotiation and setting the correct renderer on the response.

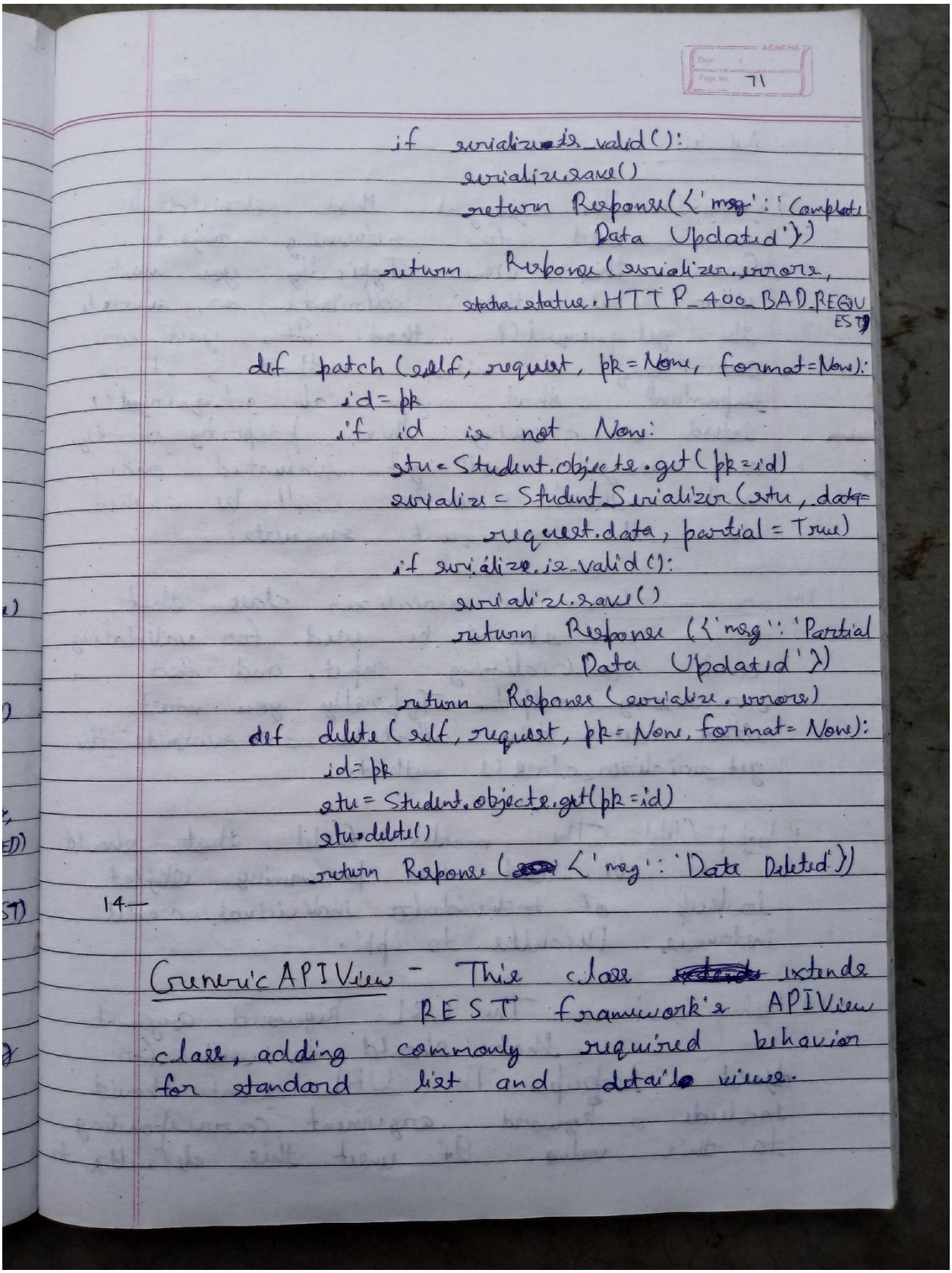
· Any APIException exceptions will be caught and mediated into appropriate responses.

· Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.



Ex- views.py

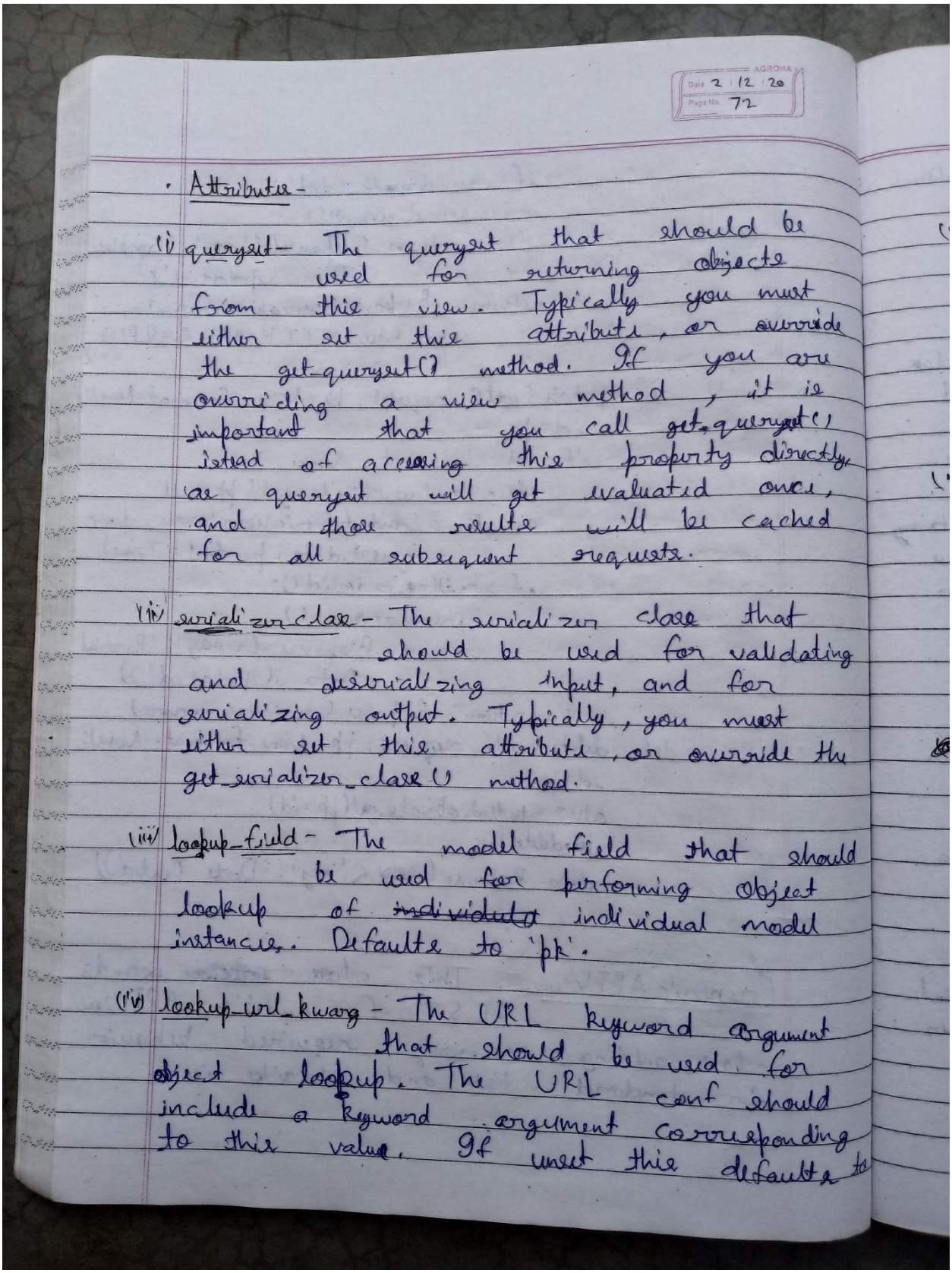
```
from rest_framework.response import Response
from .models import Student
from .serializers import StudentSerializer
from rest_framework import status
from rest_framework.views import APIView
class StudentAPI(APIView):
    def get(self, request, pk=None, format=None):
        id=pk
        if id is not None:
            stu=Student.objects.get(id=id)
            serializer=StudentSerializer(stu)
            return Response(serializer.data)
        stu=Student.objects.all()
        serializer=StudentSerializer(stu, many=True)
        return Response(serializer.data)
    def post(self, request, format=None):
        serializer=StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data Created'},
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=
                        status.HTTP_400_BAD_REQUEST)
    def put(self, request, pk=None, format=None):
        id=pk
        if id is not None:
            stu=Student.objects.get(pk=id)
            serializer=StudentSerializer(stu, data=
                                         request.data)
```



```
        if serializer.is_valid():
            serializer.save()
            return Response({'msg': 'Complete Data Updated'})
        return Response(serializer.errors,
            status.status.HTTP_400_BAD_REQUEST)

    def patch(self, request, pk=None, format=None):
        id = pk
        if id is not None:
            stu = Student.objects.get(pk=id)
            serializer = Student_Serializer(stu, data=
                request.data, partial=True)
            if serializer.is_valid():
                serializer.save()
                return Response({'msg': 'Partial
                    Data Updated'})
            return Response(serializer.errors)
    def delete(self, request, pk=None, format=None):
        id = pk
        stu = Student.objects.get(pk=id)
        stu.delete()
        return Response({'msg': 'Data Deleted'})
```

14—

Generic API View - This class extends REST framework's APIView class, adding commonly required behavior for standard list and detail views.



Date 2/12/20
Page No. 72

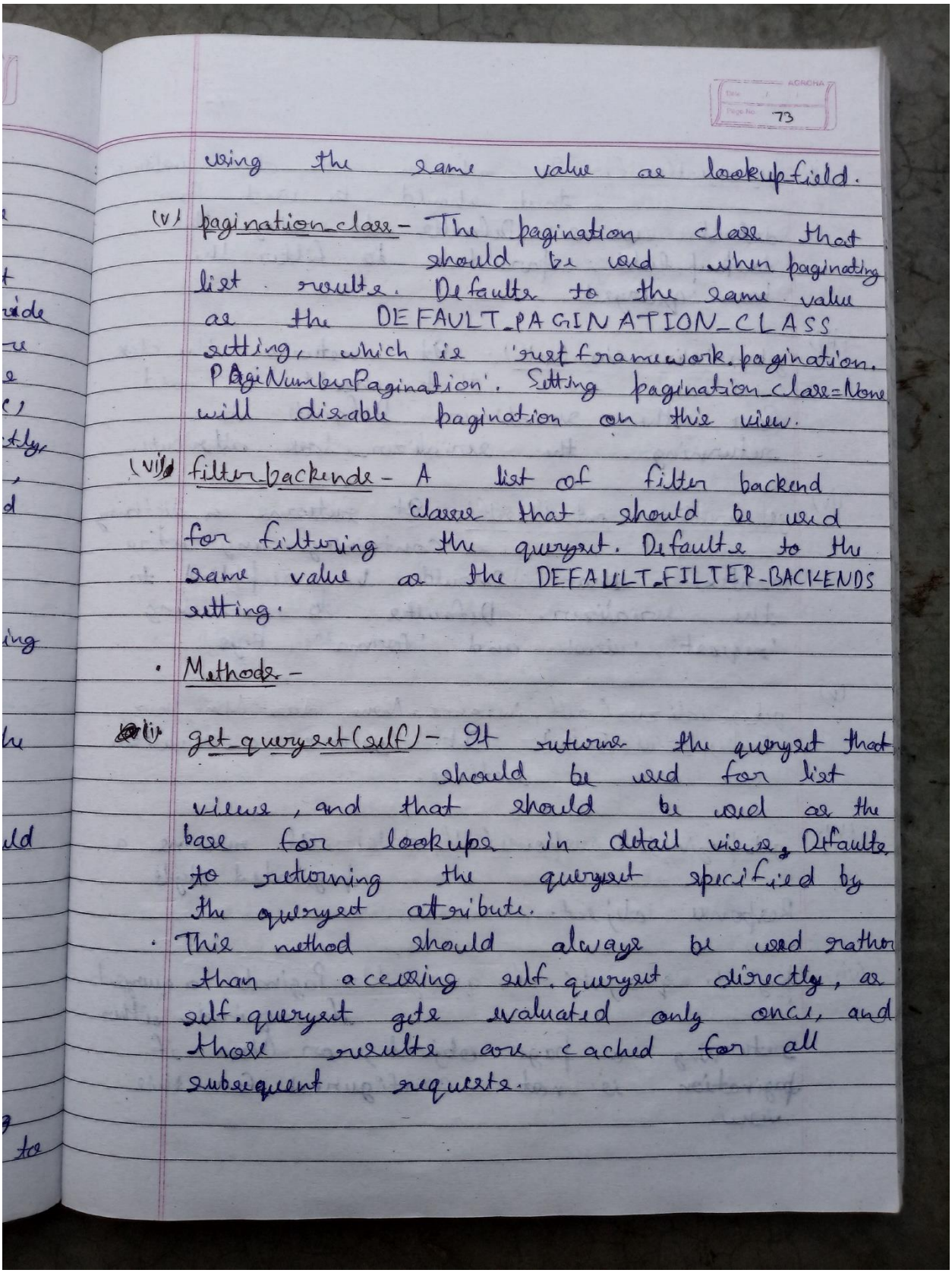
- Attributes -

(i) queryset - The queryset that should be used for returning objects from this view. Typically you must either set this attribute, or override the get_queryset() method. If you are overriding a view method, it is important that you call get_queryset() instead of accessing this property directly, as queryset will get evaluated once, and those results will be cached for all subsequent requests.

(ii) serializer_class - The serializer class that should be used for validating and deserializing input, and for serializing output. Typically, you must either set this attribute, or override the get_serializer_class() method.

(iii) lookup_field - The model field that should be used for performing object lookup of ~~individuals~~ individual model instances. Defaults to 'pk'.

(iv) lookup_url_kwarg - The URL keyword argument that should be used for object lookup. The URL conf should include a keyword argument corresponding to this value. If unset this defaults to



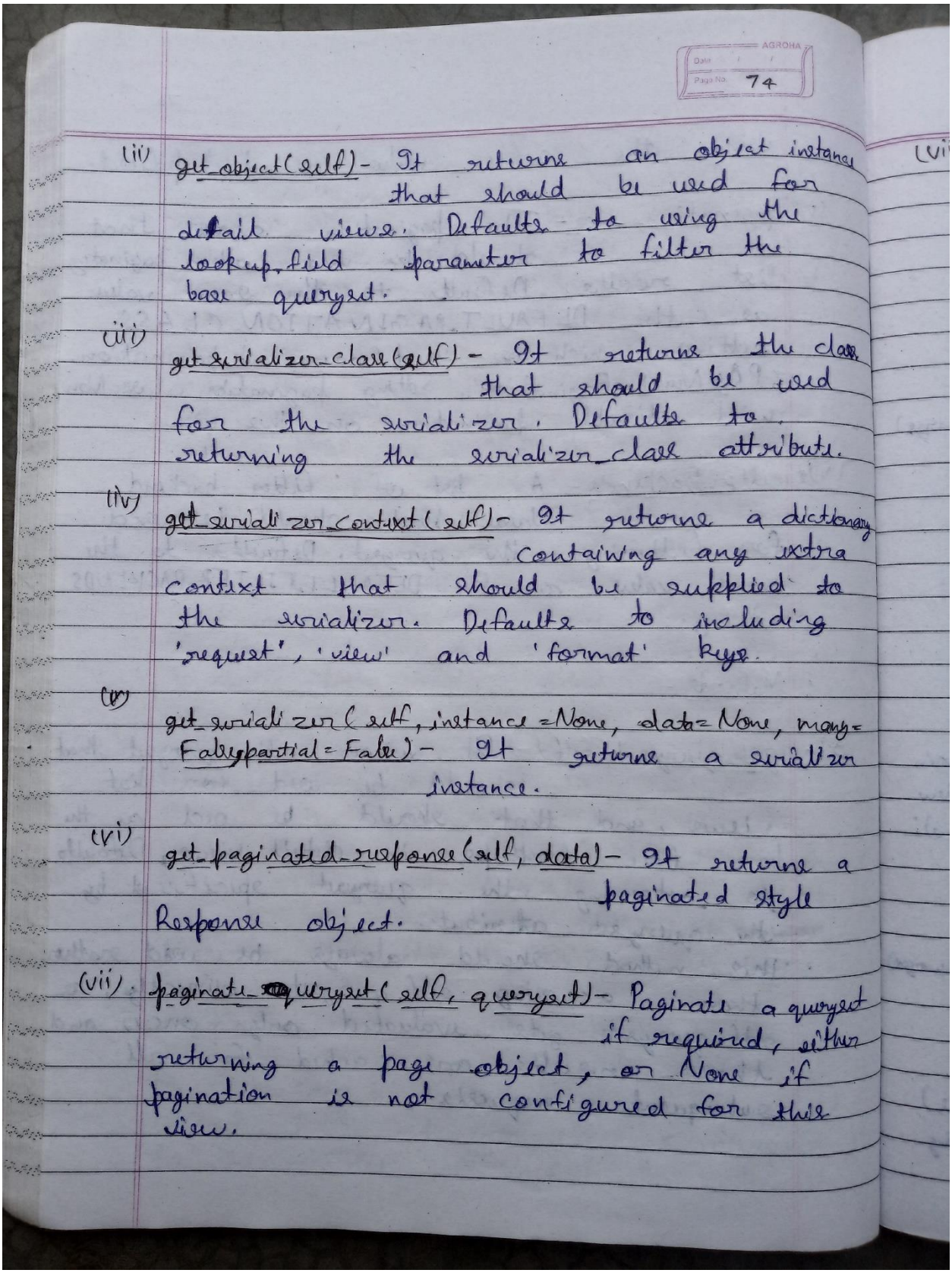
using the same value as lookup_field.

(v) pagination_class - The pagination class that should be used when paginating list results. Defaults to the same value as the DEFAULT_PAGINATION_CLASS setting, which is 'rest_framework.pagination.PageNumberPagination'. Setting pagination_class=None will disable pagination on this view.

(vi) filter_backends - A list of filter backend classes that should be used for filtering the queryset. Defaults to the same value as the DEFAULT_FILTER_BACKENDS setting.

- Methods -

(i) get_queryset(self) - It returns the queryset that should be used for list views, and that should be used as the base for lookups in detail views. Defaults to returning the queryset specified by the queryset attribute.
- This method should always be used rather than accessing self.queryset directly, as self.queryset gets evaluated only once, and those results are cached for all subsequent requests.



(ii) get_object(self) - It returns an object instance that should be used for detail views. Defaults to using the lookup_field parameter to filter the base queryset.

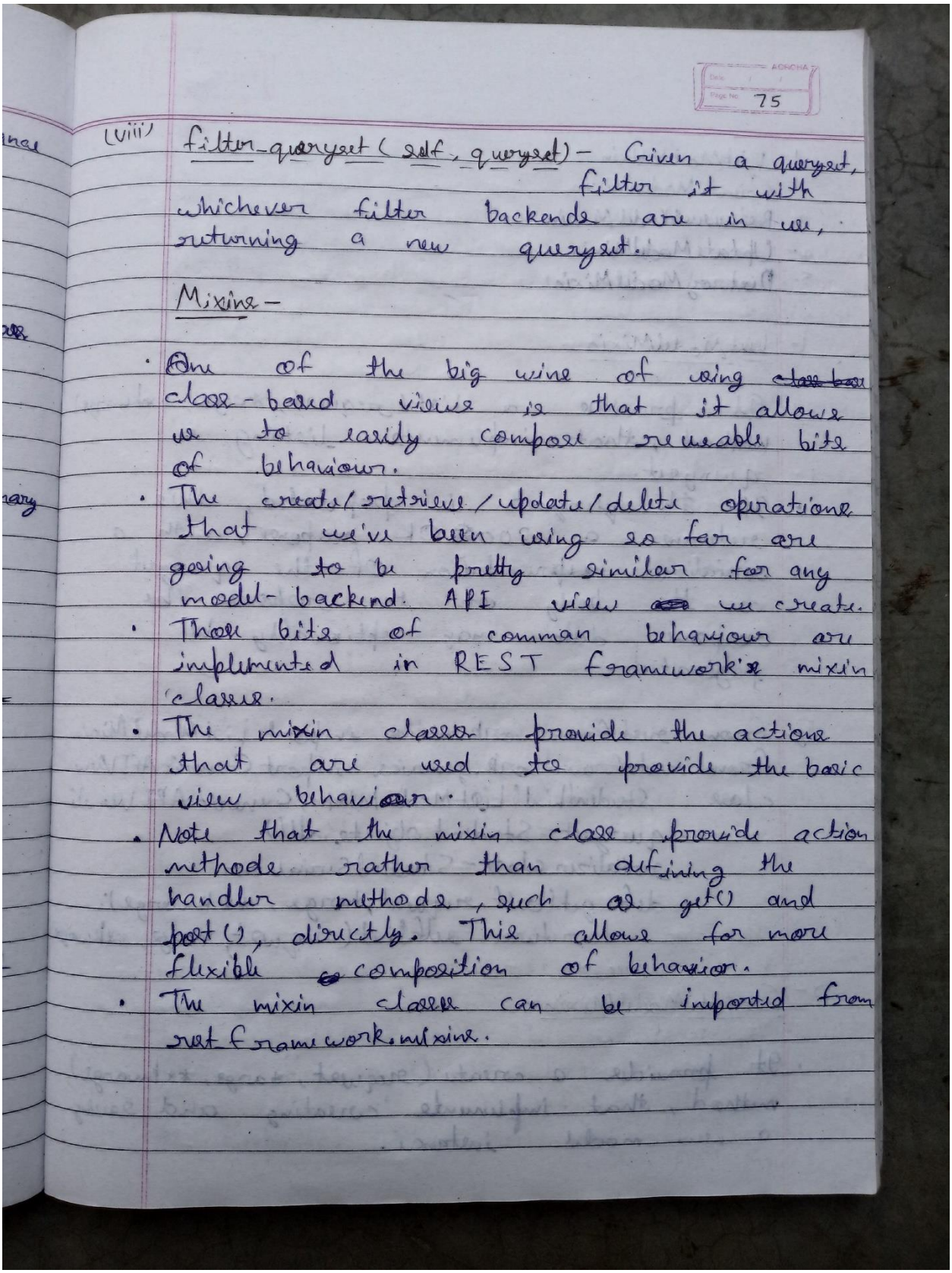
(iii) get_serializer_class(self) - It returns the class that should be used for the serializer. Defaults to returning the serializer_class attribute.

(iv) get_serializer_context(self) - It returns a dictionary containing any extra context that should be supplied to the serializer. Defaults to including 'request', 'view' and 'format' keys.

(v) get_serializer(self, instance=None, data=None, many=False, partial=False) - It returns a serializer instance.

(vi) get_paginated_response(self, data) - It returns a paginated style Response object.

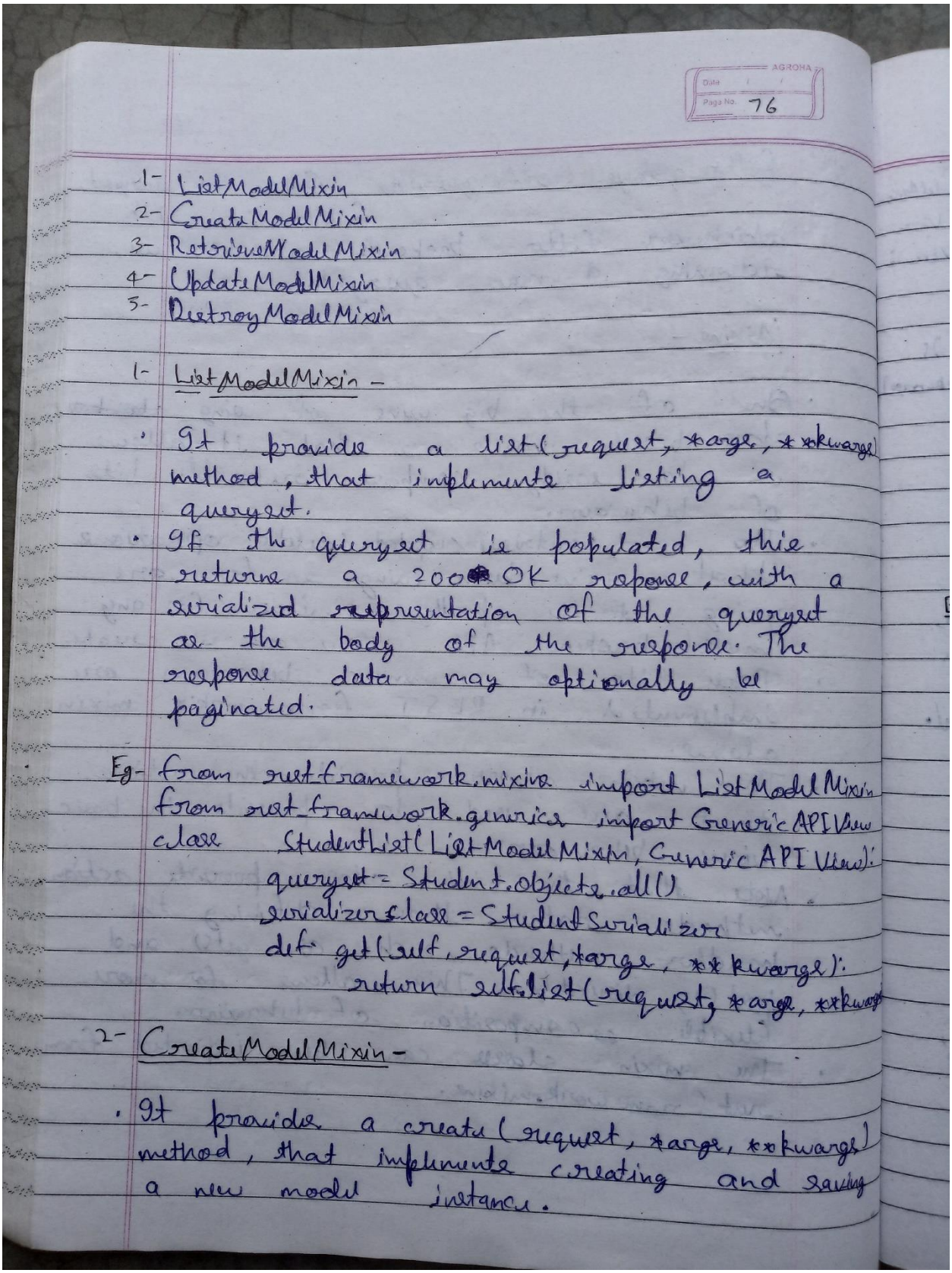
(vii) paginate_queryset(self, queryset) - Paginate a queryset if required, either returning a page object, or None if pagination is not configured for this view.



(viii) filter_queryset(self, queryset) - Given a queryset, filter it with whichever filter backends are in use, returning a new queryset.

Mixins -

- One of the big wins of using class-based views is that it allows us to easily compose reusable bits of behaviour.
- The create/retrieve/update/delete operations that we've been using so far are going to be pretty similar for any model-backend API views we create.
- Those bits of common behaviour are implemented in REST framework's mixin classes.
- The mixin classes provide the actions that are used to provide the basic view behaviour.
- Note that the mixin class provide action methods rather than defining the handler methods, such as get() and post(), directly. This allows for more flexible composition of behaviour.
- The mixin classes can be imported from rest_framework.mixins.



1- ListModelMixin
2- CreateModelMixin
3- RetrieveModelMixin
4- UpdateModelMixin
5- DestroyModelMixin

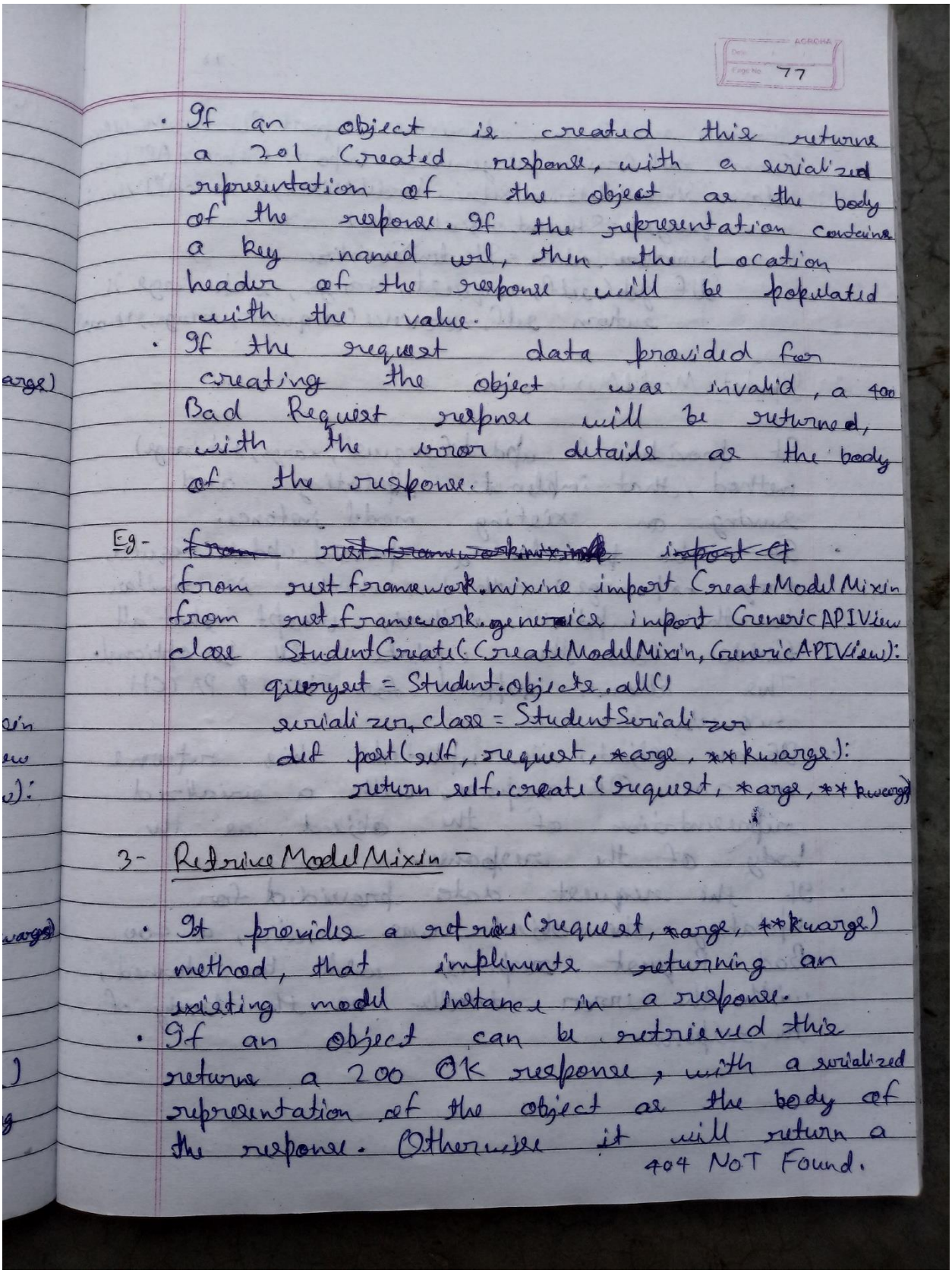
1- ListModelMixin -

- It provide a list(request, *args, **kwargs) method, that implements listing a queryset.
- If the queryset is populated, this returns a 200 OK response, with a serialized representation of the queryset as the body of the response. The response data may optionally be paginated.

Eg- from rest_framework.mixins import ListModelMixin
from rest_framework.generics import GenericAPIView
class StudentList(ListModelMixin, GenericAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer
def get(self, request, *args, **kwargs):
return self.list(request, *args, **kwargs

2- CreateModelMixin -

- It provides a create(request, *args, **kwargs) method, that implements creating and saving a new model instance.

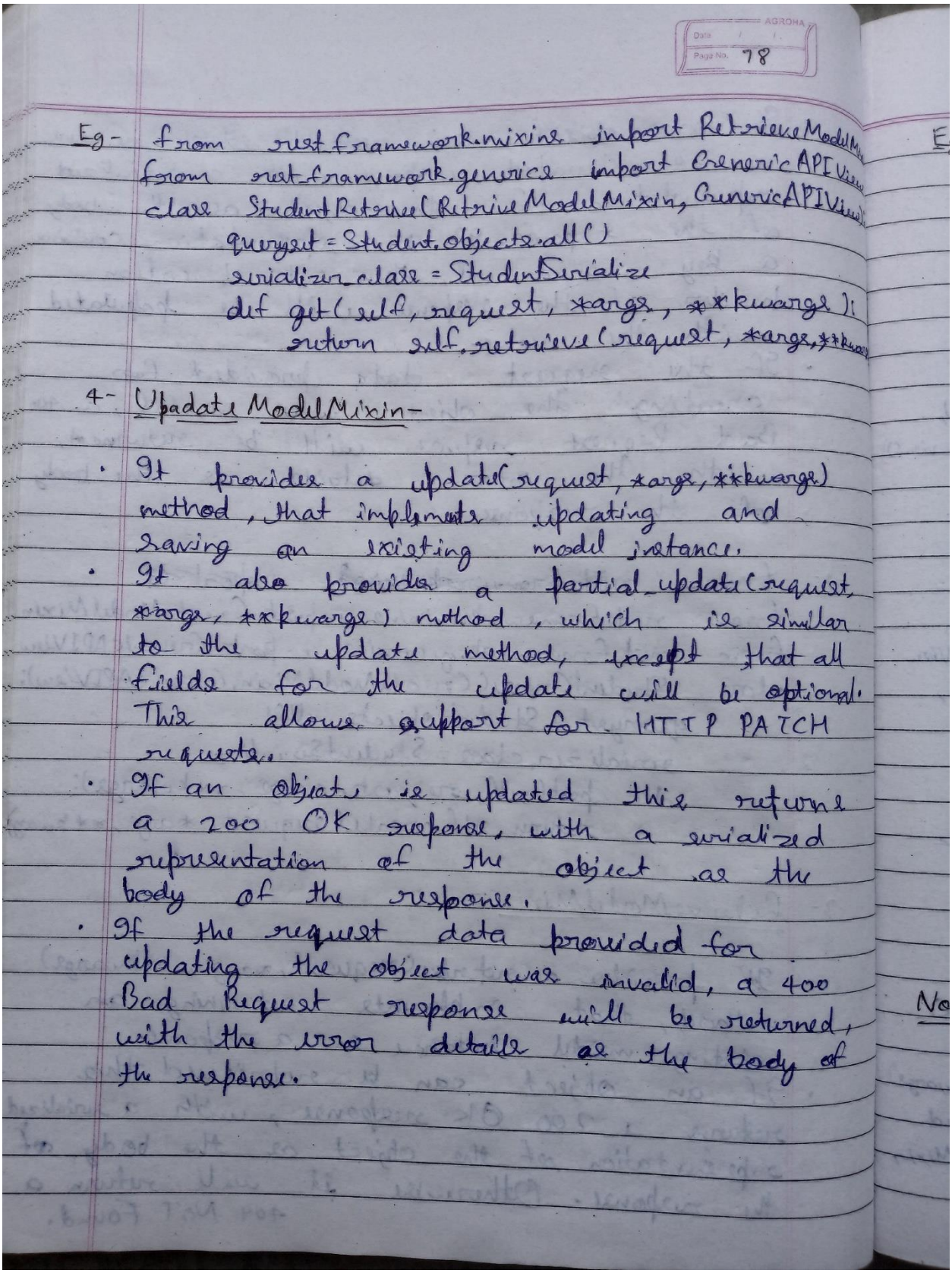


- If an object is created this returns a 201 Created response, with a serialized representation of the object as the body of the response. If the representation contains a key named url, then the Location header of the response will be populated with the value.
- If the request data provided for creating the object was invalid, a 400 Bad Request response will be returned, with the error details as the body of the response.

Eg- ~~from rest_framework.mixins import~~ ~~CR~~
from rest_framework.mixins import CreateModelMixin
from rest_framework.generics import GenericAPIView
class StudentCreate(CreateModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

3- RetriveModelMixin -

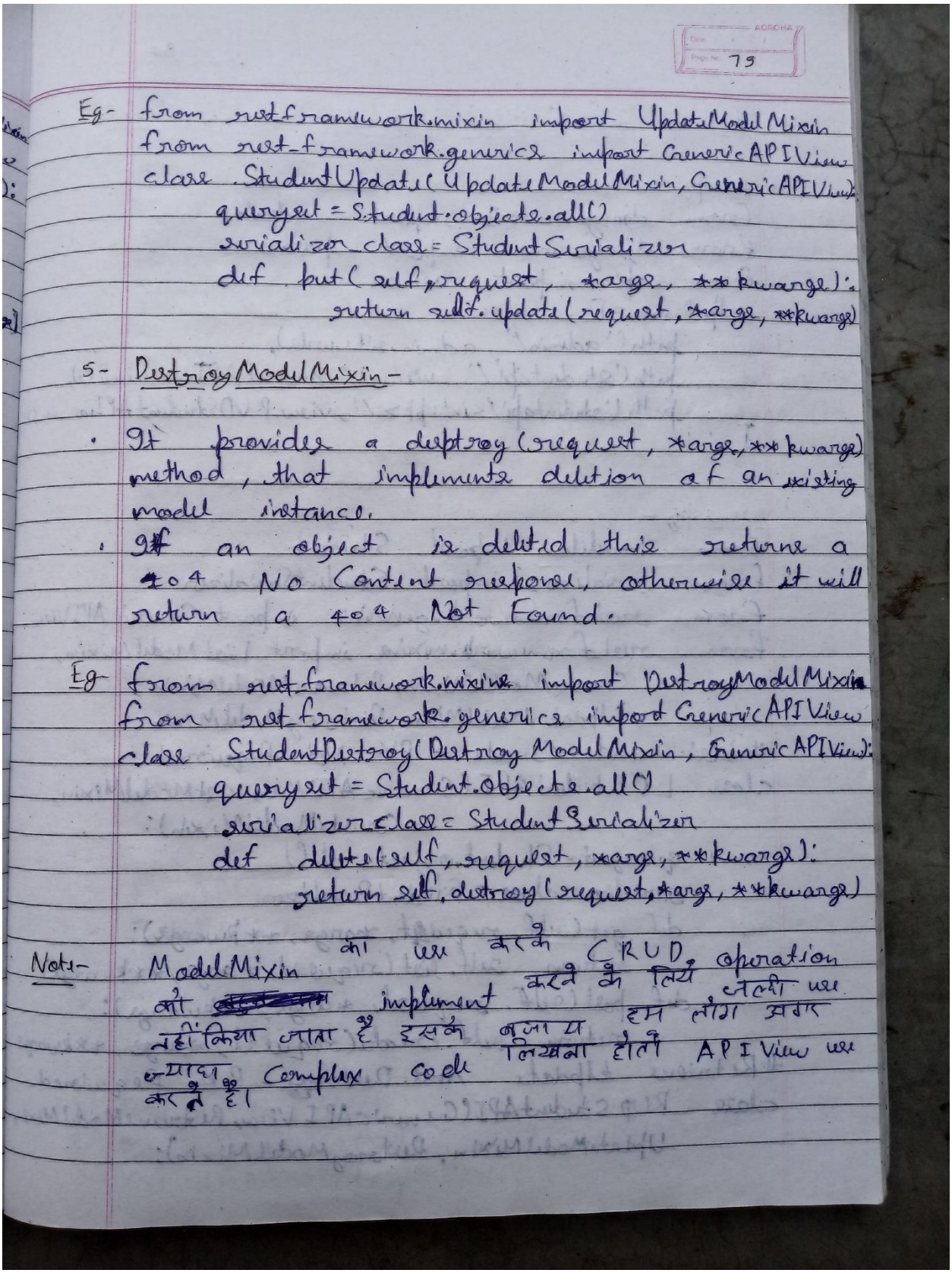
- It provides a retrive(request, *args, **kwargs) method, that implements returning an existing model instance in a response.
- If an object can be retrieved this returns a 200 OK response, with a serialized representation of the object as the body of the response. Otherwise it will return a 404 NOT Found.



Eg- from rest_framework.mixins import RetrieveModelMixin
from rest_framework.generics import GenericAPIView
class StudentRetrieve(RetriveModelMixin, GenericAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

4- Updated ModelMixin-

- It provides a update(request, *args, **kwargs) method, that implements updating and saving an existing model instance.
- It also provides a partial_update(request, *args, **kwargs) method, which is similar to the update method, except that all fields for the update will be optional. This allows support for HTTP PATCH requests.
- If an object is updated this returns a 200 OK response, with a serialized representation of the object as the body of the response.
- If the request data provided for updating the object was invalid, a 400 Bad Request response will be returned, with the error details as the body of the response.



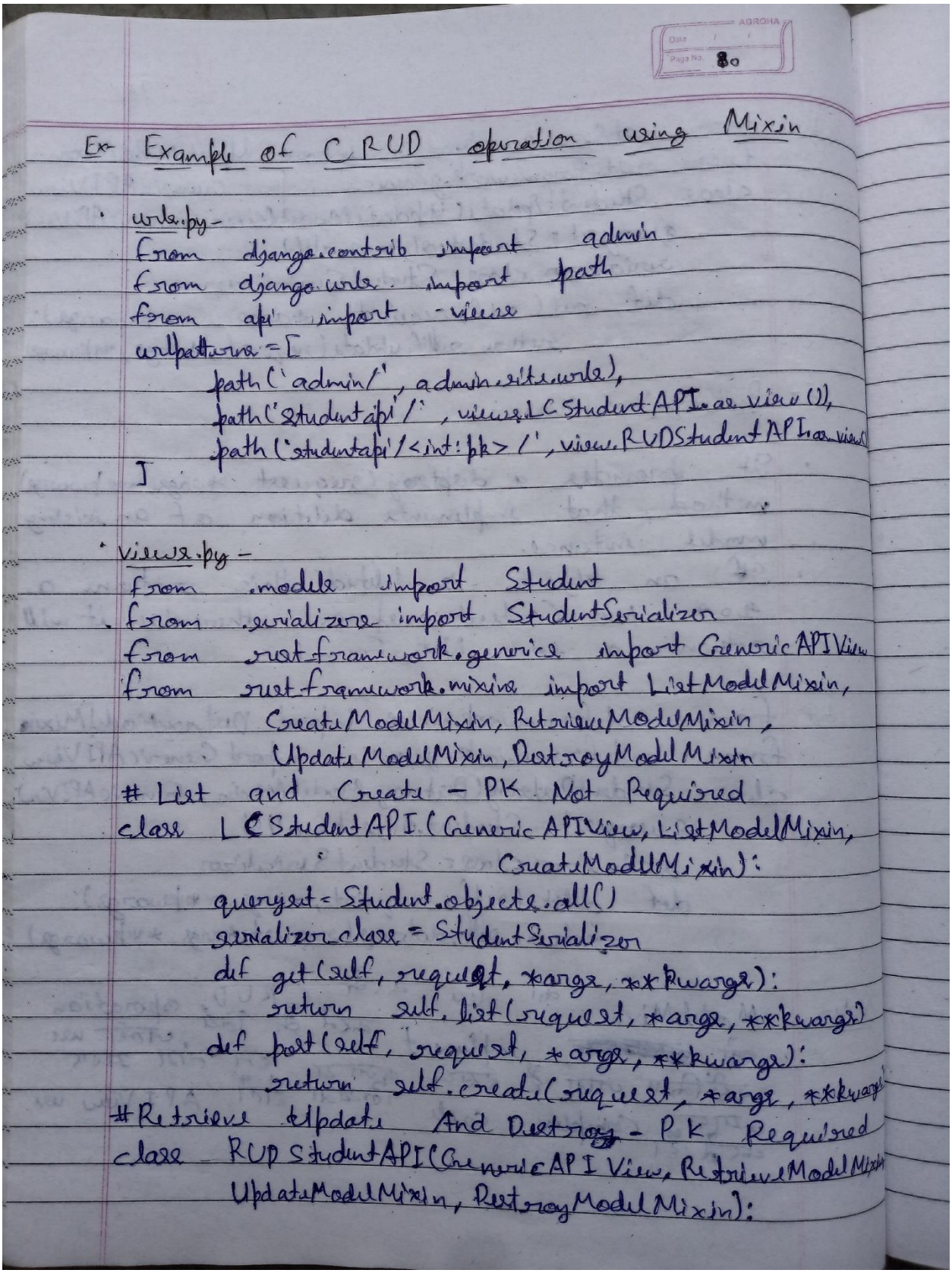
Eg- from rest_framework.mixin import UpdateModelMixin
from rest_framework.generics import GenericAPIView
class StudentUpdate(UpdateModelMixin, GenericAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer
def put(self, request, *args, **kwargs):
return self.update(request, *args, **kwargs)

5- DestroyModelMixin -

- It provides a destroy(request, *args, **kwargs) method, that implements deletion of an existing model instance.
- If an object is deleted this returns a 204 No Content response, otherwise it will return a 404 Not Found.

Eg- from rest_framework.mixins import DestroyModelMixin
from rest_framework.generics import GenericAPIView
class StudentDestroy(DestroyModelMixin, GenericAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer
def delete(self, request, *args, **kwargs):
return self.destroy(request, *args, **kwargs)

Note- ModelMixin का use करके CRUD operation को implement करने के लिये तभी use नहीं किया जाता है इसके बजाय हम लोग अगर ज्यादा Complex code लिखना होता API View use करते है।



Ex- Example of CRUD operation using Mixin

- urls.py -

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/', views.LCStudentAPI.as_view()),
    path('studentapi/<int:pk>/', views.RUDStudentAPI.as_view()),
]
```

- views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import GenericAPIView
from rest_framework.mixins import ListModelMixin,
        CreateModelMixin, RetrieveModelMixin,
        UpdateModelMixin, DestroyModelMixin
# List and Create - PK Not Required
class LCStudentAPI(GenericAPIView, ListModelMixin,
                   CreateModelMixin):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
# Retrieve Update And Destroy - PK Required
class RUDStudentAPI(GenericAPIView, RetrieveModelMixin,
        UpdateModelMixin, DestroyModelMixin):
```

```
queryset = Student.object.all()
serializer_class = StudentSerializer
def get(self, request, *args, **kwargs):
    return self.retrieve(request, *args, **
                                    kwargs)
def put(self, request, *args, **kwargs):
    return self.update(request, *args, **k
                                    wargs)
def delete(self, request, *args, **kwargs):
    return self.destroy(request, *args, **
                                    kwargs)
```

15 —

Concrete View Class -

- The following classes are the concrete generic views.
- If you're using generic views this is normally the level you'll be working at unless you need heavily customized behavior.
- The view classes can be imported from rest_framework.generics —

1- ListAPIView
2- CreateAPIView
3- RetrieveAPIView
4- UpdateAPIView
5- DestroyAPIView
6- ListCreateAPIView
7- RetrieveUpdateAPIView
8- RetrieveDestroyAPIView
9- RetrieveUpdateDestroyAPIView

1- ListAPIView -

- It is used for read-only endpoints to represent a collection of model instances. It provides a get method handler.
- Extends - GenericAPIView, ListModelMixin

Eg- from rest_framework.generics import ListAPIView
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer

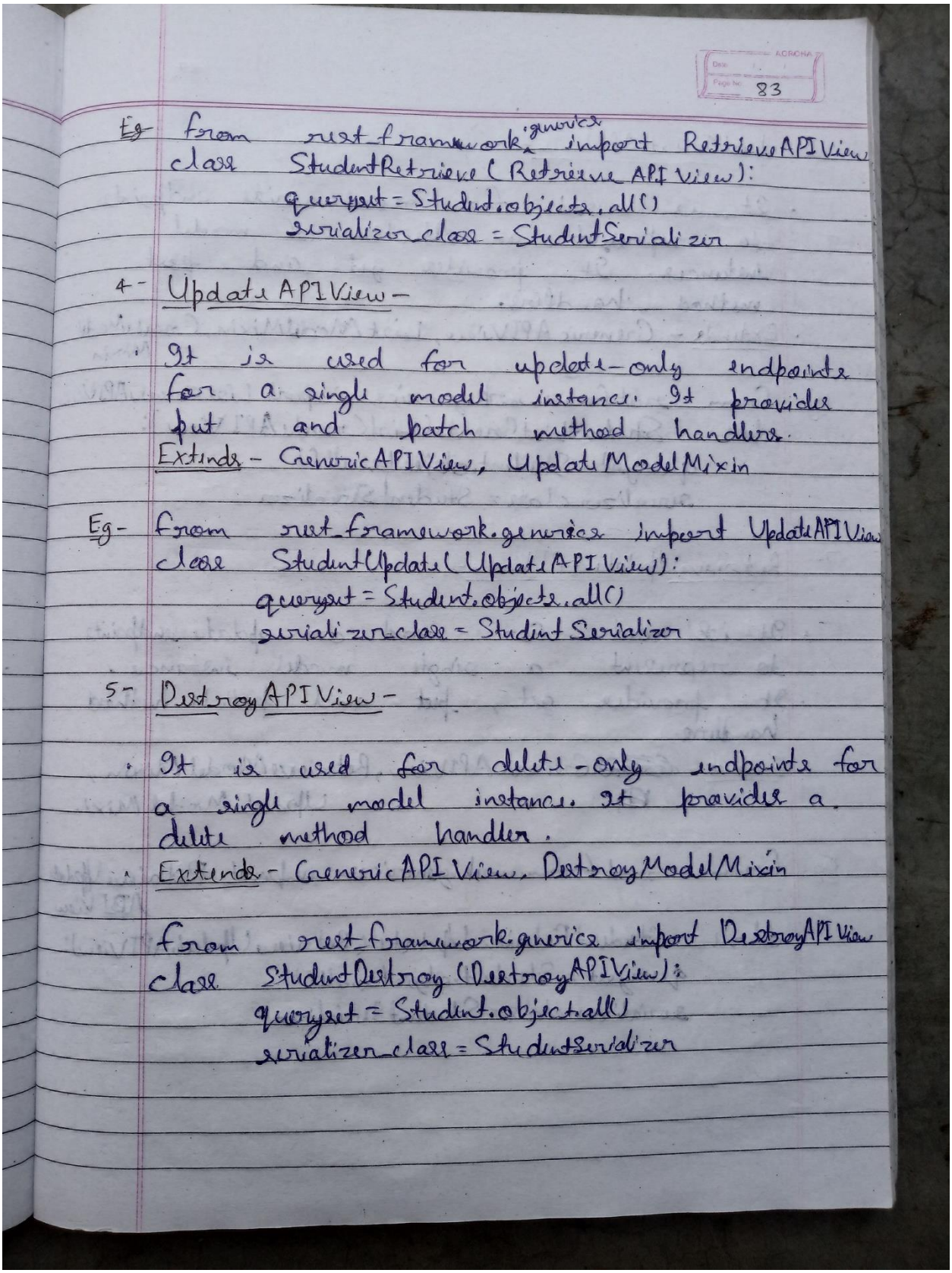
2- CreateAPIView -

- It is used for create-only endpoints.
- It provides post method handler.
- Extends - GenericAPIView, CreateModelMixin

Eg- from rest_framework.generics import CreateAPIView
class StudentCreate(CreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer.

3- RetrieveAPIView -

- It is used for read-only endpoints to represent a single model instance. It provides a get method handler.
- Extends - GenericAPIView, RetrieveModelMixin



Eg- from rest_framework.generics import RetrieveAPIView
class StudentRetrieve(RetrieveAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer

4- Update API View -

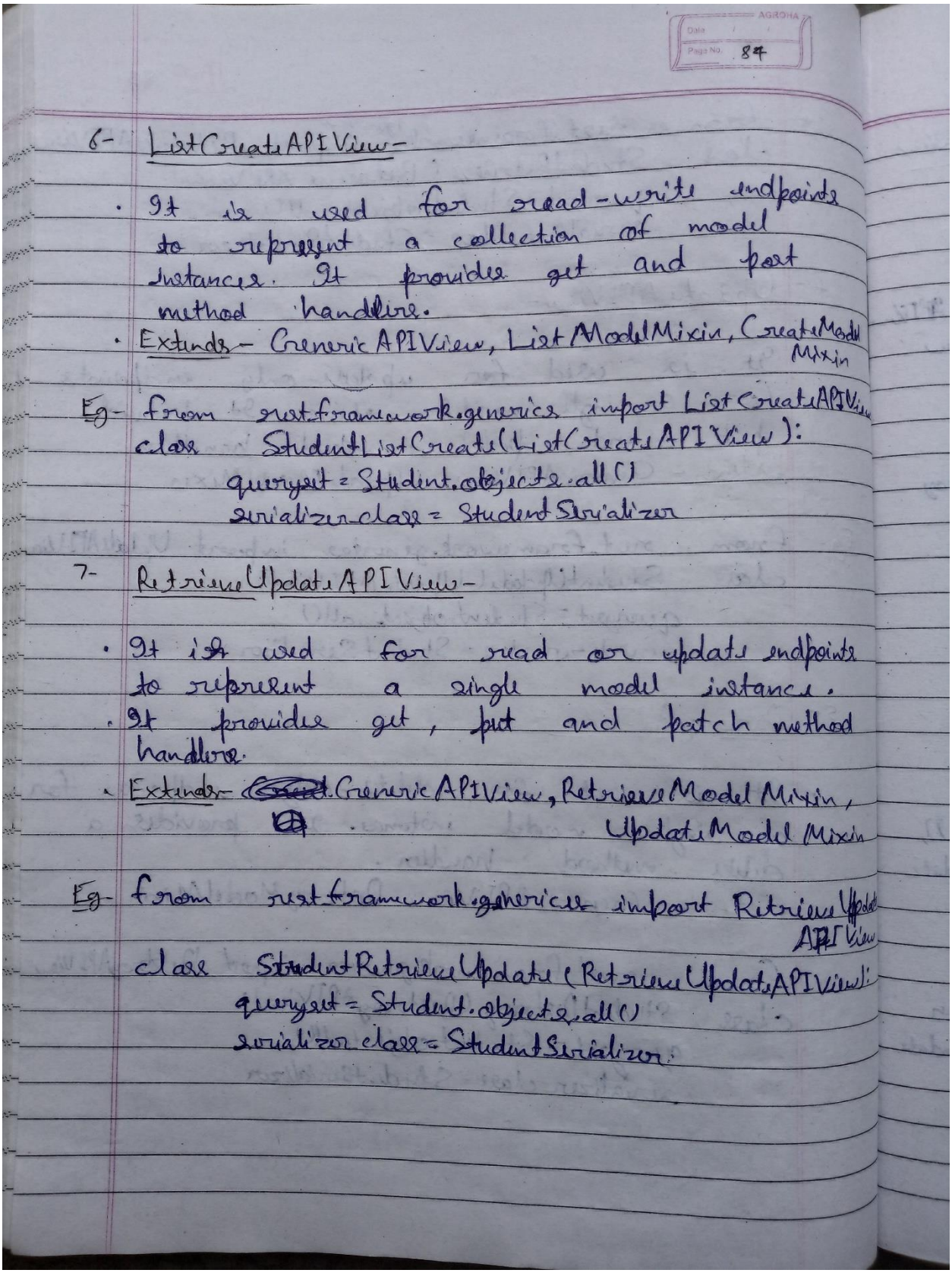
- It is used for update-only endpoints for a single model instance. It provides put and patch method handlers.
- Extends - GenericAPIView, UpdateModelMixin

Eg- from rest_framework.generics import UpdateAPIView
class StudentUpdate(UpdateAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer

5- Destroy API View -

- It is used for delete-only endpoints for a single model instance. It provides a delete method handler.
- Extends - GenericAPIView, DestroyModelMixin

from rest_framework.generics import DestroyAPIView
class StudentDestroy(DestroyAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer



6- ListCreateAPIView-

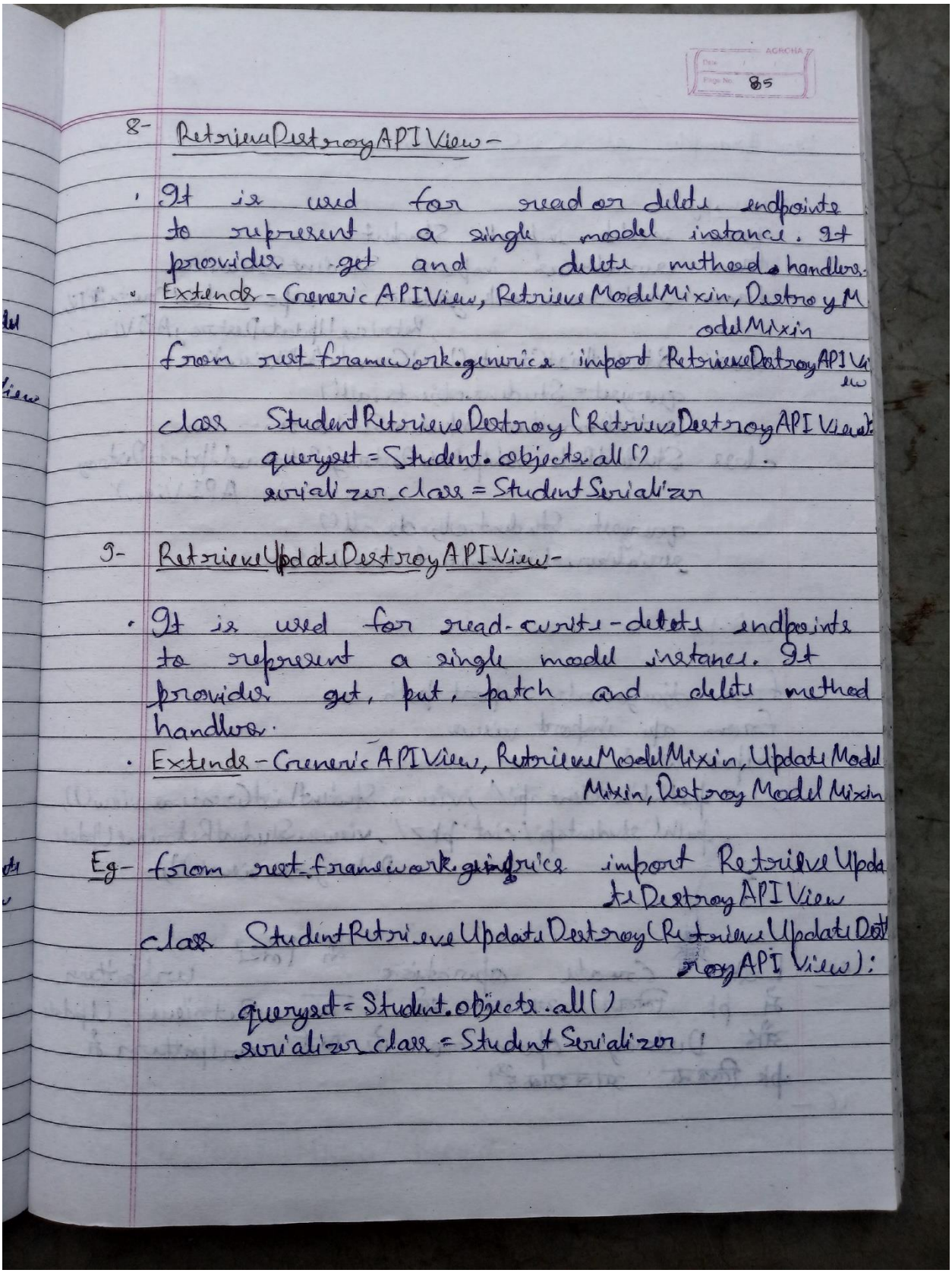
- It is used for read-write endpoints to represent a collection of model instances. It provides get and post method handlers.
- Extends- GenericAPIView, ListModelMixin, CreateModelMixin

Eg- from rest_framework.generics import ListCreateAPIView
class StudentListCreate(ListCreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer

7- RetrieveUpdateAPIView-

- It is used for read or update endpoints to represent a single model instance.
- It provides get, put and patch method handlers.
- Extends- GenericAPIView, RetrieveModelMixin, UpdateModelMixin

Eg- from rest_framework.generics import RetrieveUpdateAPIView
class StudentRetrieveUpdate(RetrieveUpdateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer



8- RetrieveDestroyAPIView-

- It is used for read or delete endpoints to represent a single model instance. It provides get and delete method handlers.
- Extends - GenericAPIView, RetrieveModelMixin, DestroyModelMixin

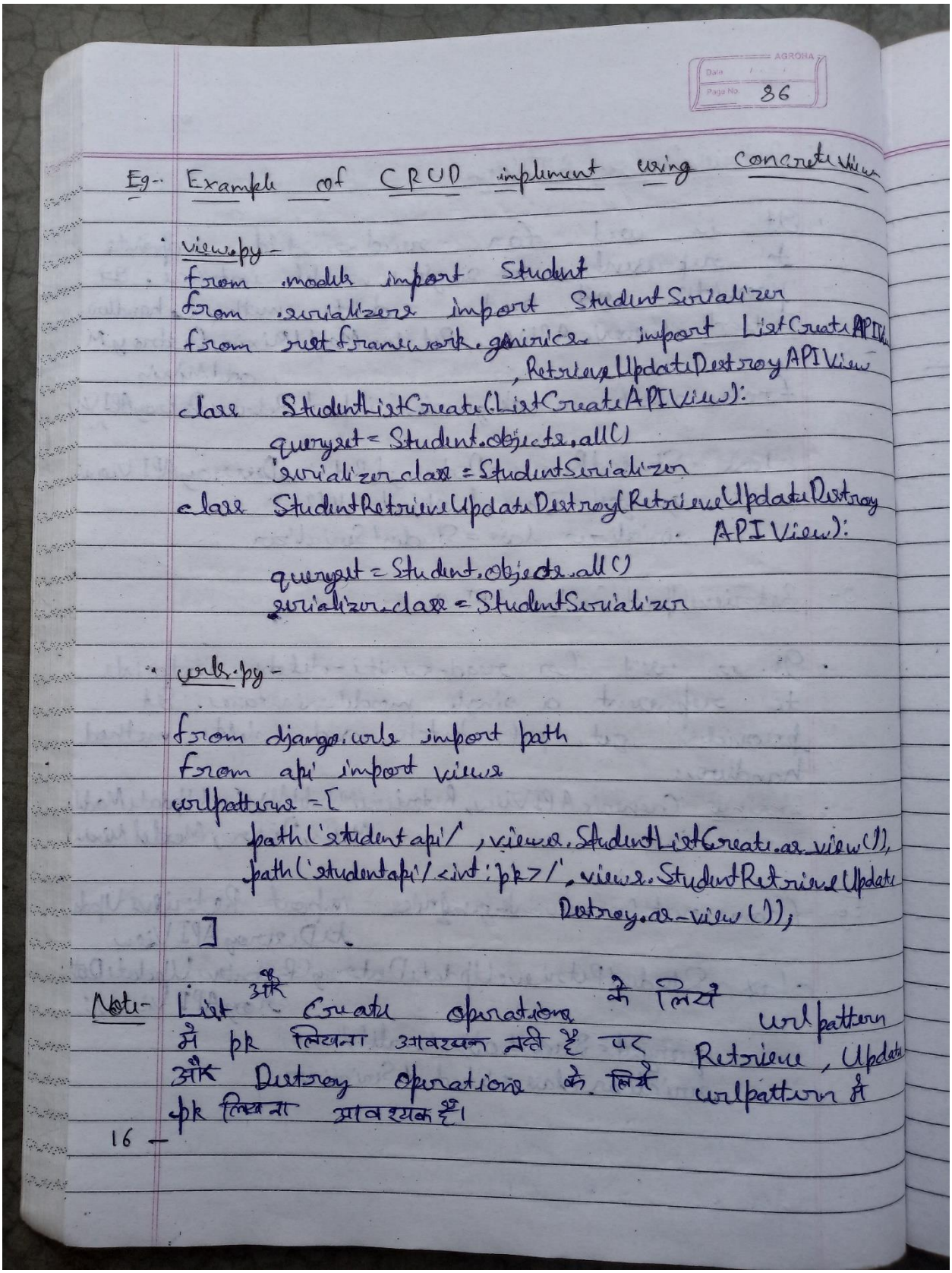
```
from rest_framework.generics import RetrieveDestroyAPIView

class StudentRetrieveDestroy(RetrieveDestroyAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

9- RetrieveUpdateDestroyAPIView-

- It is used for read-write-delete endpoints to represent a single model instance. It provides get, put, patch and delete method handlers.
- Extends - GenericAPIView, RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin

Eg-

```
from rest_framework.generics import RetrieveUpdateDestroyAPIView

class StudentRetrieveUpdateDestroy(RetrieveUpdateDestroyAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```



Eg- Example of CRUD implement using concrete view

- views.py -

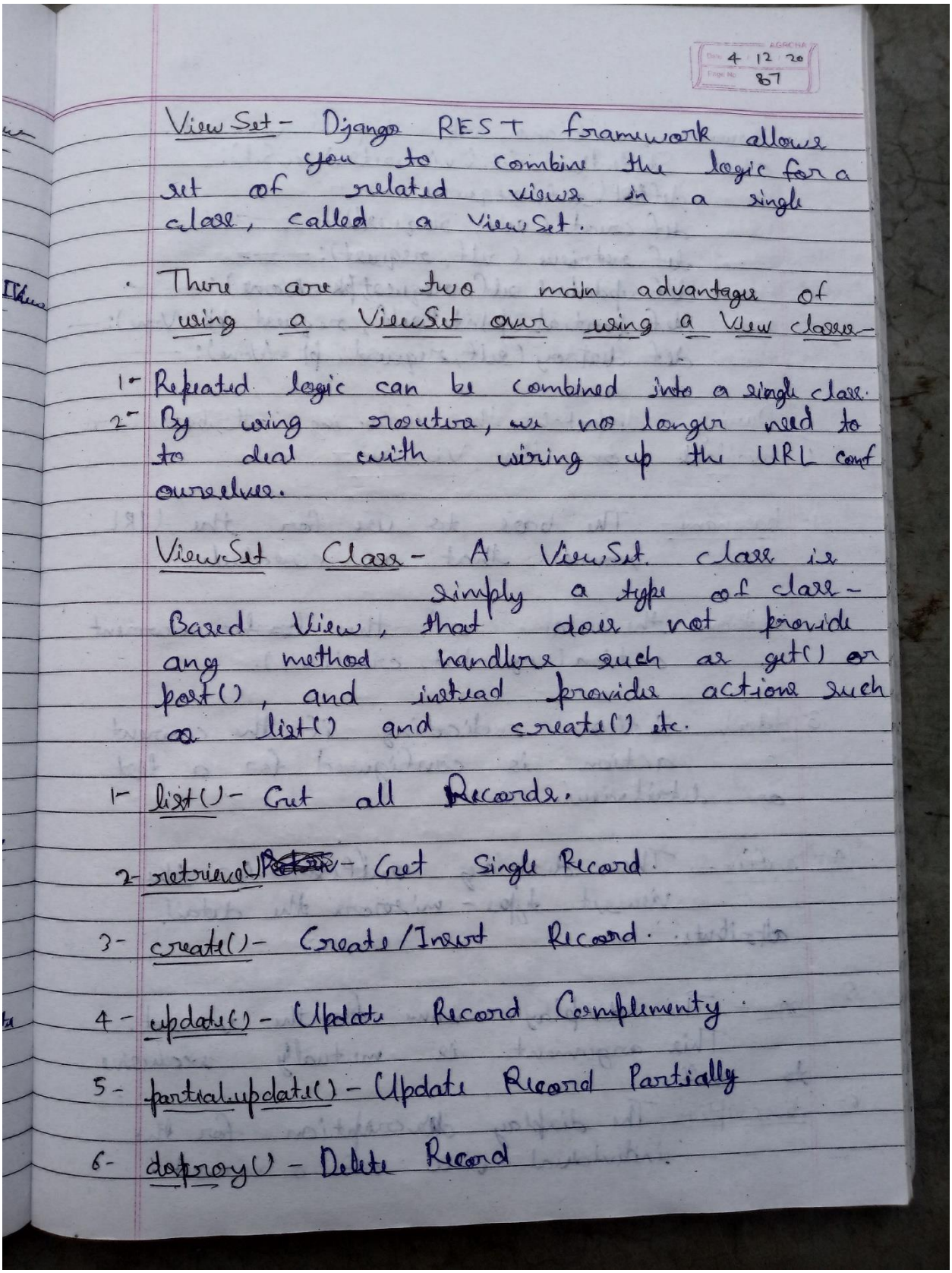
```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListCreateAPIView
                    , RetrieveUpdateDestroyAPIView
class StudentListCreate(ListCreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
class StudentRetrieveUpdateDestroy(RetrieveUpdateDestroy
                                    APIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

- urls.py -

```
from django.urls import path
from api import views
urlpatterns = [
    path('studentapi/', views.StudentListCreate.as_view()),
    path('studentapi/<int:pk>/', views.StudentRetrieveUpdate
                                    Destroy.as_view()),
]
```

Note- List और Create operations के लिये urlpattern में pk लिखना आवश्यक नहीं है पर Retrieve, Update और Destroy operations के लिये urlpattern में pk लिखना आवश्यक है।

16 -



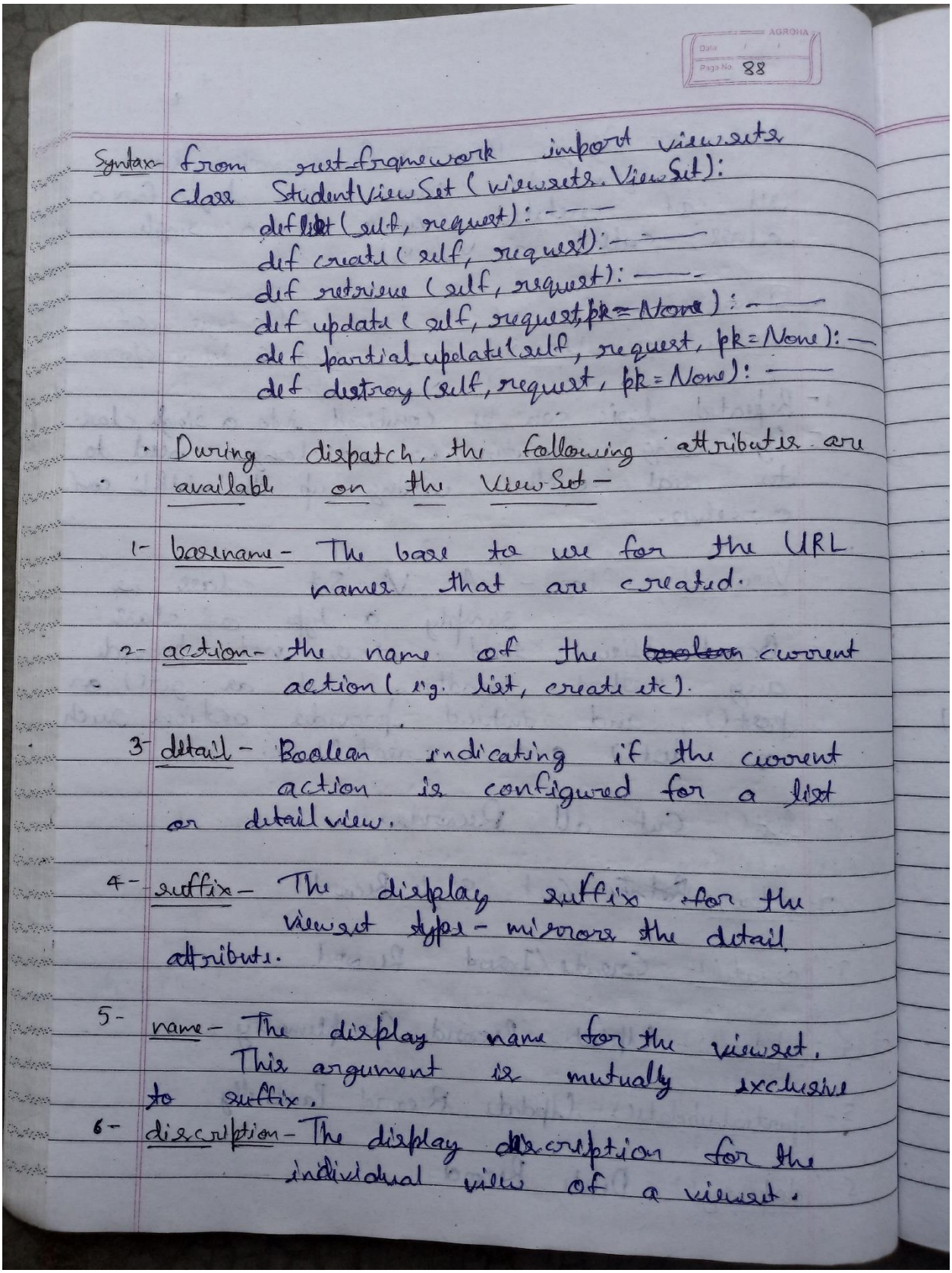
ViewSet - Django REST framework allows you to combine the logic for a set of related views in a single class, called a ViewSet.

- There are two main advantages of using a ViewSet over using a View class -

1- Repeated logic can be combined into a single class.
2- By using routers, we no longer need to to deal with wiring up the URL conf ourselves.

ViewSet Class - A ViewSet class is simply a type of class-Based View, that does not provide any method handlers such as get() or post(), and instead provides actions such as list() and create() etc.

1- list() - Get all Records.
2- retrieve() - Get Single Record.
3- create() - Create/Insert Record.
4- update() - Update Record Complementy.
5- partial_update() - Update Record Partially
6- destroy() - Delete Record



Syntax- from rest_framework import viewsets
class StudentViewSet(viewsets.ViewSet):
    def list(self, request): ———
    def create(self, request): ———
    def retrieve(self, request): ———
    def update(self, request, pk=None): ———
    def partial_update(self, request, pk=None): —
    def destroy(self, request, pk=None): ———

- During dispatch, the following attributes are available on the ViewSet -

1- basename - The base to use for the URL names that are created.

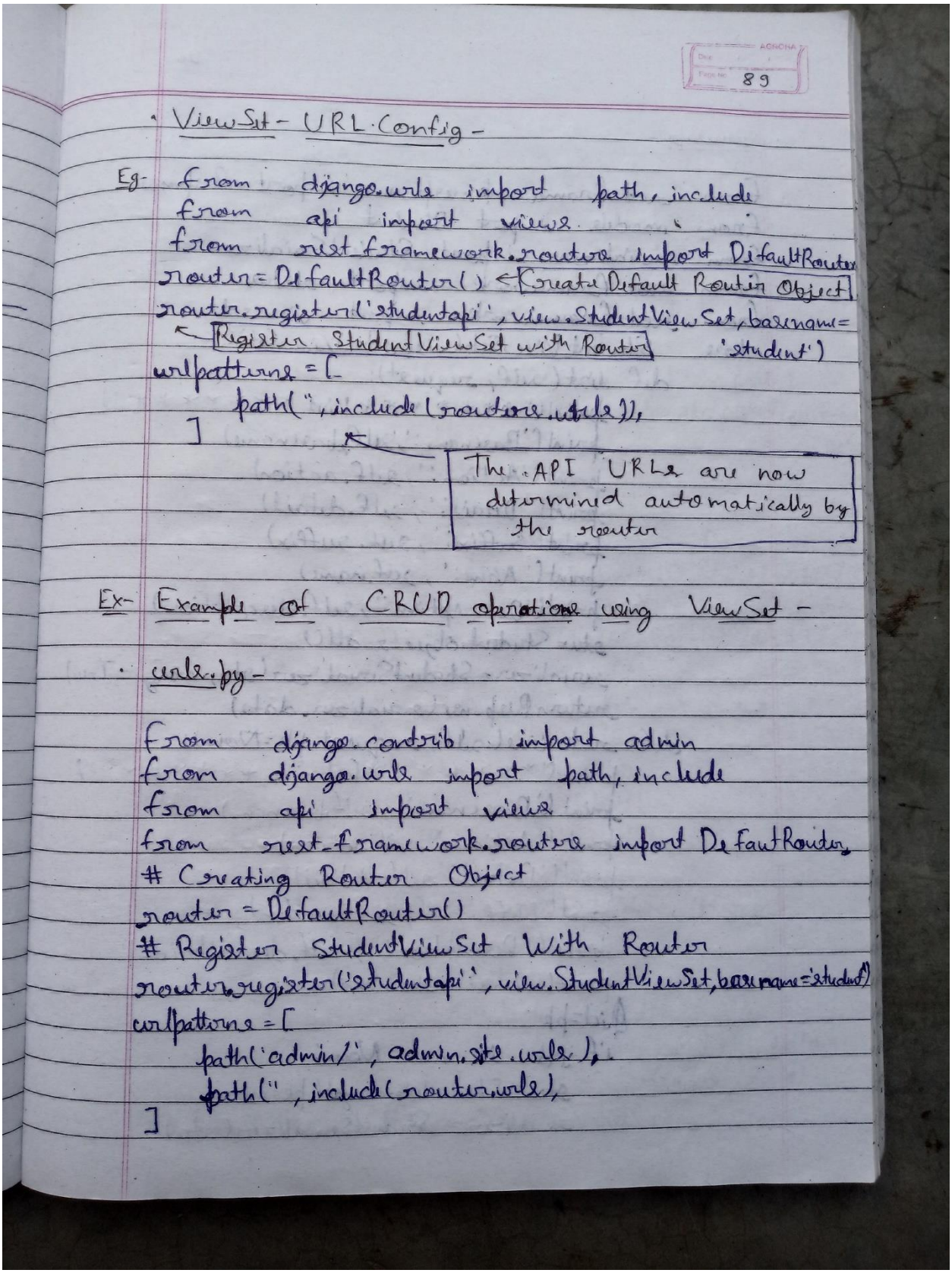
2- action - the name of the ~~boolean~~ current action (e.g. list, create etc).

3- detail - Boolean indicating if the current action is configured for a list or detail view.

4- suffix - The display suffix for the viewset type - mirrors the detail attribute.

5- name - The display name for the viewset. This argument is mutually exclusive to suffix.

6- discription - The display description for the individual view of a viewset.



Page No. 89

- ViewSet - URL Config -

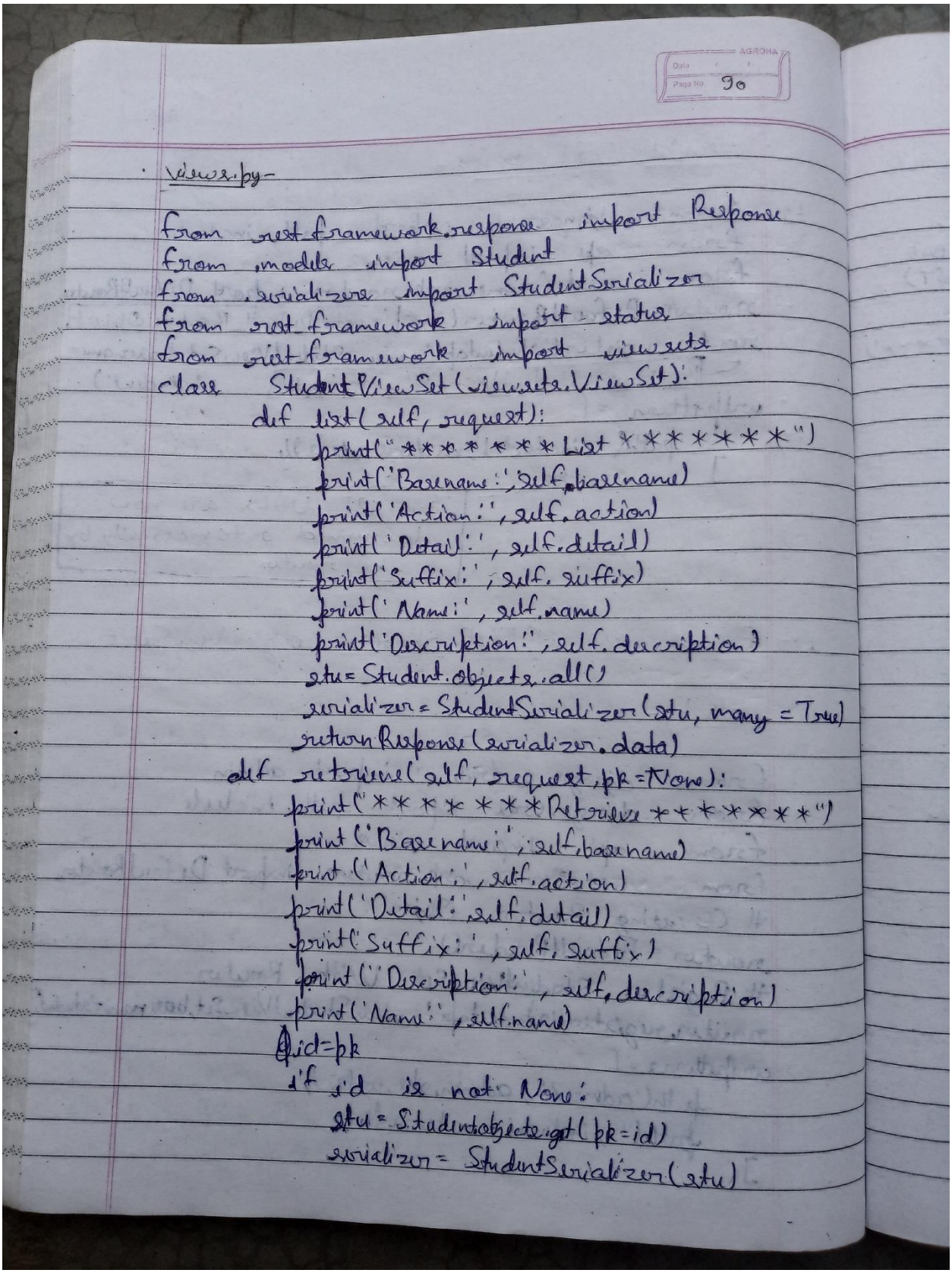
Eg-
```
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
router = DefaultRouter()          <- Create Default Router Object
router.register('studentapi', views.StudentViewSet, basename=
                                  'student')
                                  <- Register StudentViewSet with Router
urlpatterns = [
    path('', include(router.urls)),
]
```

The API URLs are now determined automatically by the router

Ex- Example of CRUD operations using ViewSet -

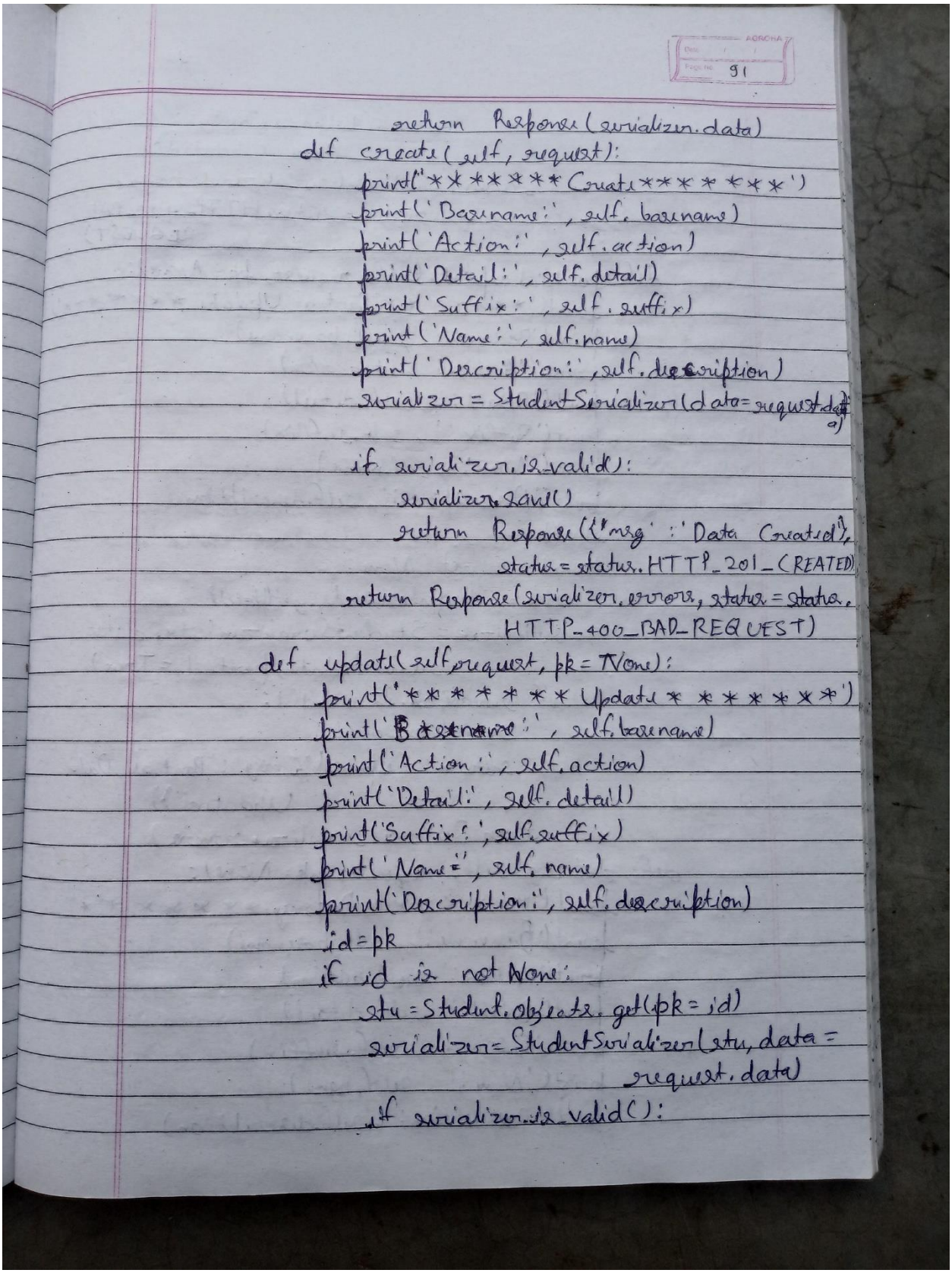
- urls.py -

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
# Creating Router Object
router = DefaultRouter()
# Register StudentViewSet With Router
router.register('studentapi', views.StudentViewSet, basename='student')
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
]
```

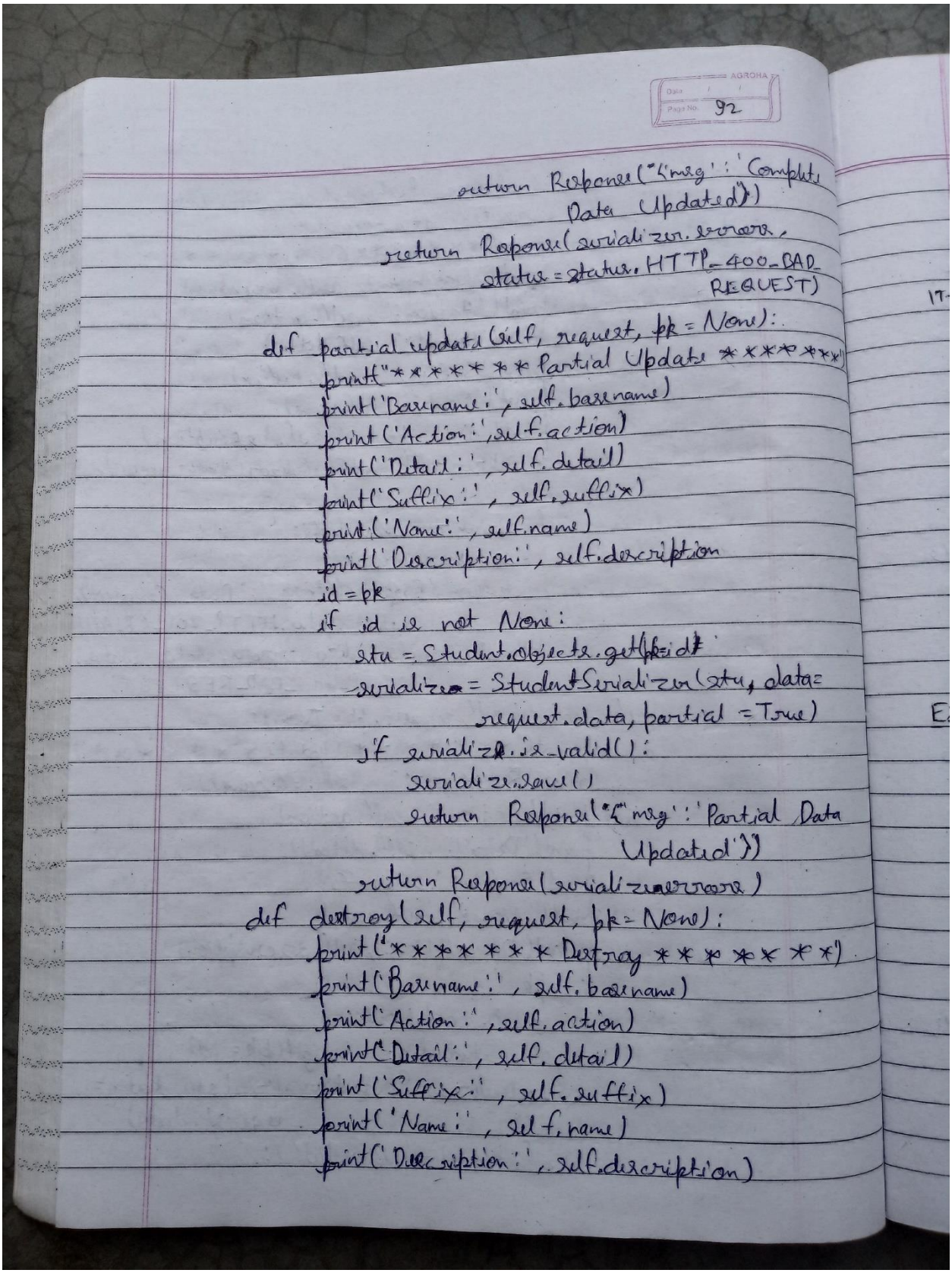


views.py -

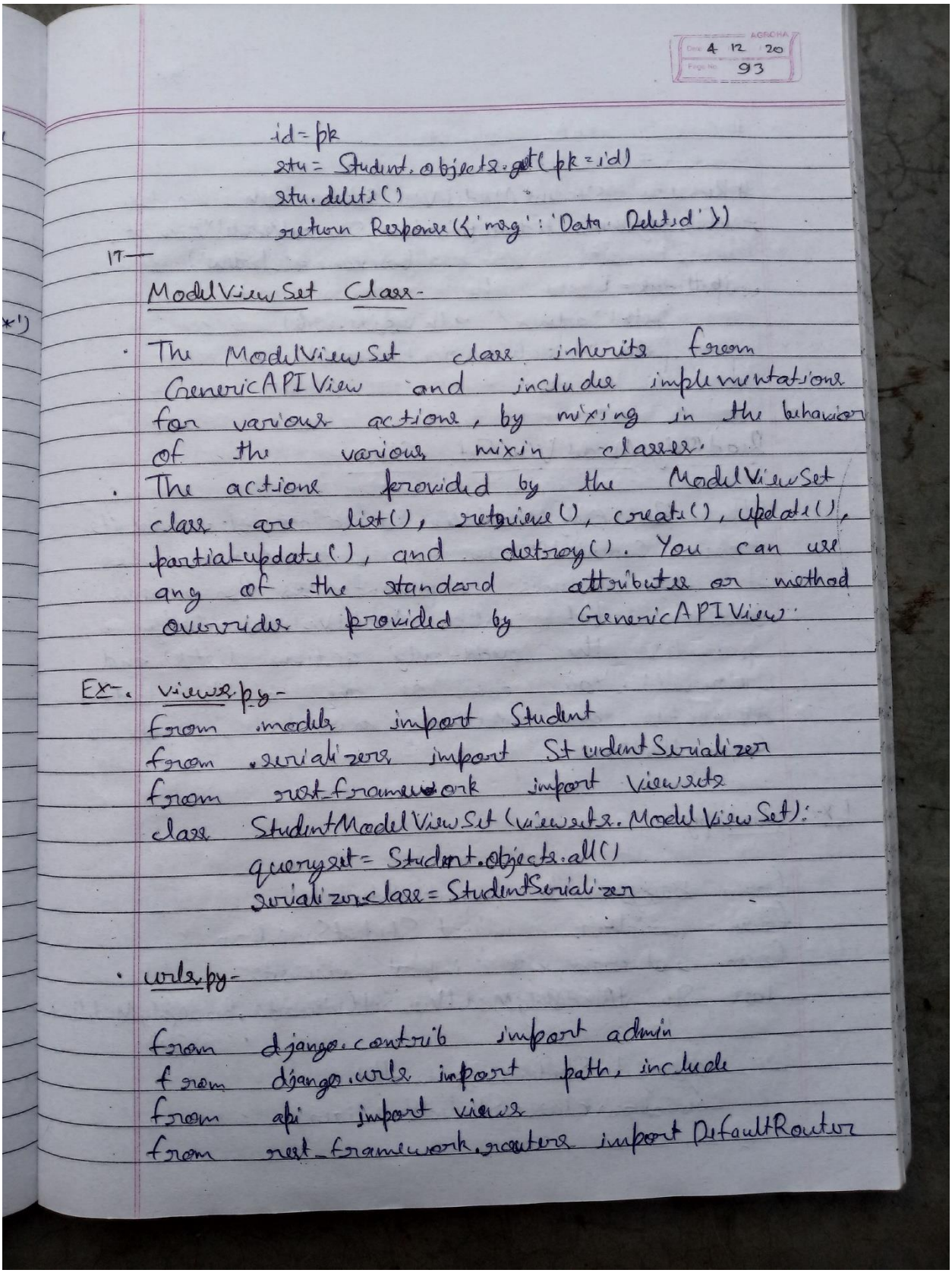
```
from rest_framework.response import Response
from .models import Student
from .serializers import StudentSerializer
from rest_framework import status
from rest_framework import viewsets
class StudentViewSet(viewsets.ViewSet):
    def list(self, request):
        print("********List *******")
        print('Basename:', self.basename)
        print('Action:', self.action)
        print('Detail:', self.detail)
        print('Suffix:', self.suffix)
        print('Name:', self.name)
        print('Description:', self.description)
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many = True)
        return Response(serializer.data)
    def retrieve(self, request, pk=None):
        print("********Retrieve ********")
        print('Basename:', self.basename)
        print('Action:', self.action)
        print('Detail:', self.detail)
        print('Suffix:', self.suffix)
        print('Description:', self.description)
        print('Name:', self.name)
        id = pk
        if id is not None:
            stu = Student.objects.get(pk=id)
            serializer = StudentSerializer(stu)
```



```
        return Response(serializer.data)
    def create(self, request):
        print('*******Create********')
        print('Basename:', self.basename)
        print('Action:', self.action)
        print('Detail:', self.detail)
        print('Suffix:', self.suffix)
        print('Name:', self.name)
        print('Description:', self.description)
        serializer = StudentSerializer(data=request.data)

        if serializer.is_valid():
            serializer.save()
            return Response({'msg': 'Data Created'},
                status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.
            HTTP_400_BAD_REQUEST)
    def update(self,request, pk=None):
        print('******** Update ********')
        print('Basename:', self.basename)
        print('Action:', self.action)
        print('Detail:', self.detail)
        print('Suffix:', self.suffix)
        print('Name:', self.name)
        print('Description:', self.description)
        id=pk
        if id is not None:
            stu=Student.objects.get(pk=id)
            serializer=StudentSerializer(stu, data=
                request.data)
            if serializer.is_valid():
```



```
        return Response({'msg': 'Complete
                Data Updated'})
    return Response(serializer.errors,
        status = status.HTTP_400_BAD_
                REQUEST)

def partial_update(self, request, pk = None):
    print("******* Partial Update ********")
    print('Basename:', self.basename)
    print('Action:', self.action)
    print('Detail:', self.detail)
    print('Suffix:', self.suffix)
    print('Name:', self.name)
    print('Description:', self.description
    id = pk
    if id is not None:
        stu = Student.objects.get(pk=id)
        serializer = StudentSerializer(stu, data=
                request.data, partial = True)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg': 'Partial Data
                    Updated'})
        return Response(serializer.errors)
def destroy(self, request, pk = None):
    print('******* Destroy ********')
    print('Basename:', self.basename)
    print('Action:', self.action)
    print('Detail:', self.detail)
    print('Suffix:', self.suffix)
    print('Name:', self.name)
    print('Description:', self.description)
```



```
    id = pk
    stu = Student.objects.get(pk = id)
    stu.delete()
    return Response({'msg': 'Data Deleted'})
```

17—

ModelViewSet Class-

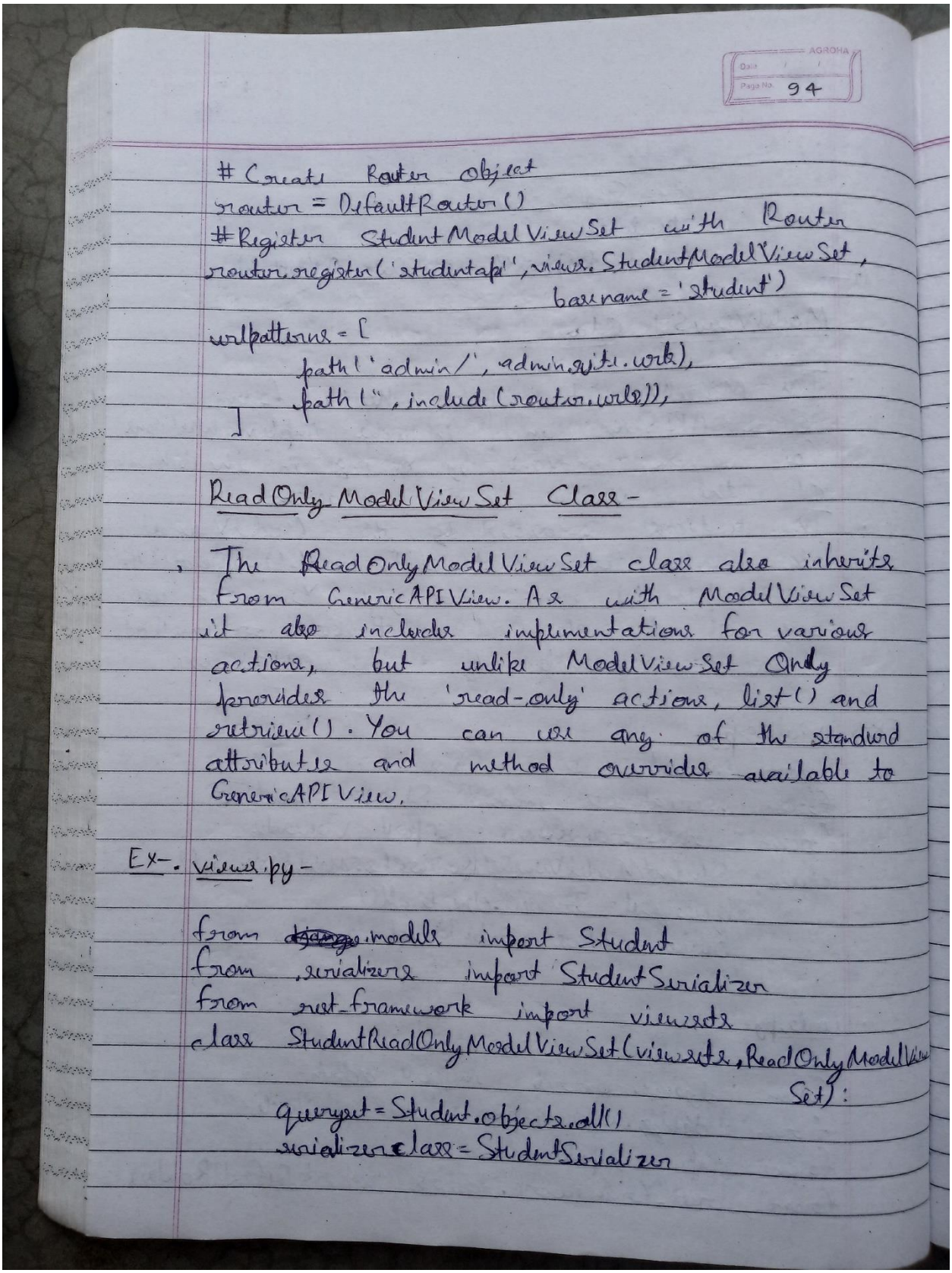
- The ModelViewSet class inherits from GenericAPIView and includes implementations for various actions, by mixing in the behavior of the various mixin classes.
- The actions provided by the ModelViewSet class are list(), retrieve(), create(), update(), partial_update(), and destroy(). You can use any of the standard attributes or method overrides provided by GenericAPIView.

Ex- views.py-

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

- urls.py-

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
```



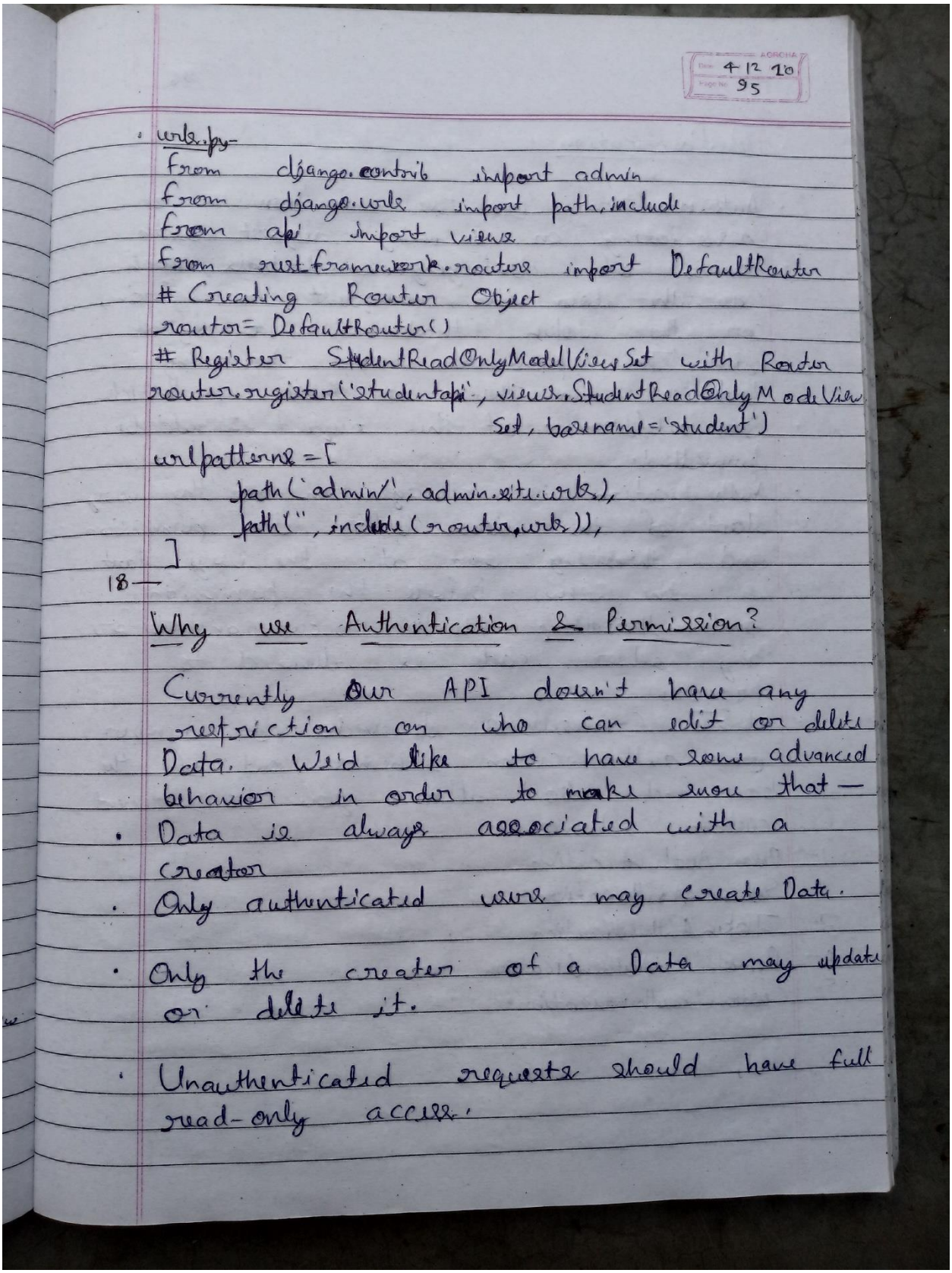
```
# Create Router object
router = DefaultRouter()
# Register StudentModelViewSet with Router
router.register('studentapi', views.StudentModelViewSet,
                basename='student')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
]
```

ReadOnlyModelViewSet Class -

- The ReadOnlyModelViewSet class also inherits from GenericAPIView. As with ModelViewSet it also includes implementations for various actions, but unlike ModelViewSet Only provides the 'read-only' actions, list() and retrieve(). You can use any of the standard attributes and method overrides available to GenericAPIView.

Ex- views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
class StudentReadOnlyModelViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```



urls.py-

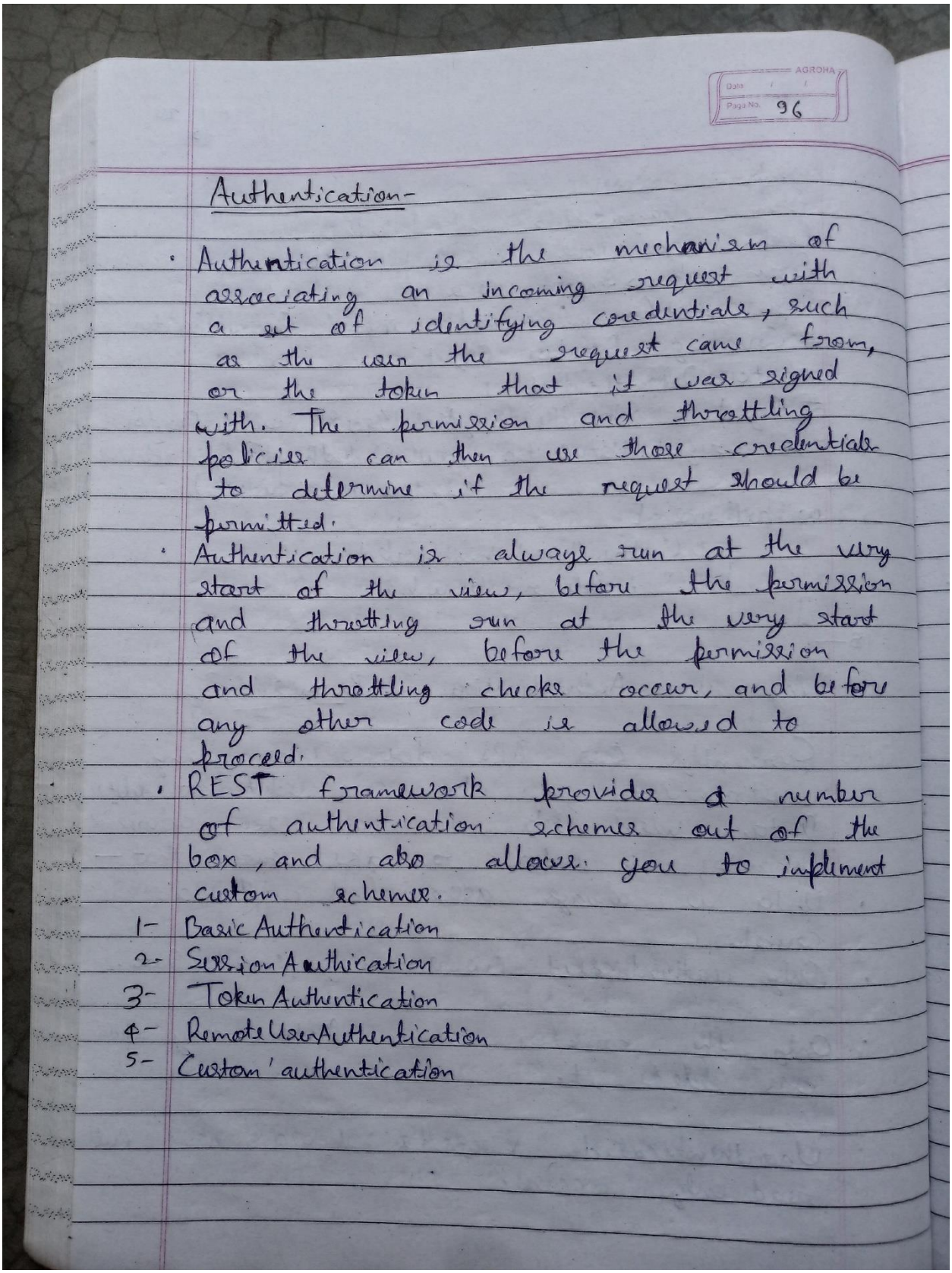
```
from django.contrib import admin
from django.urls import path,include
from api import views
from rest_framework.routers import DefaultRouter
# Creating Router Object
router= DefaultRouter()
# Register StudentReadOnlyModelViewSet with Router
router.register('studentapi', views.StudentReadOnlyModelViewSet, basename='student')
urlpatterns =[
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
]
```

18—

Why use Authentication & Permission?

Currently our API doesn't have any restriction on who can edit or delete Data. We'd like to have some advanced behavior in order to make sure that —

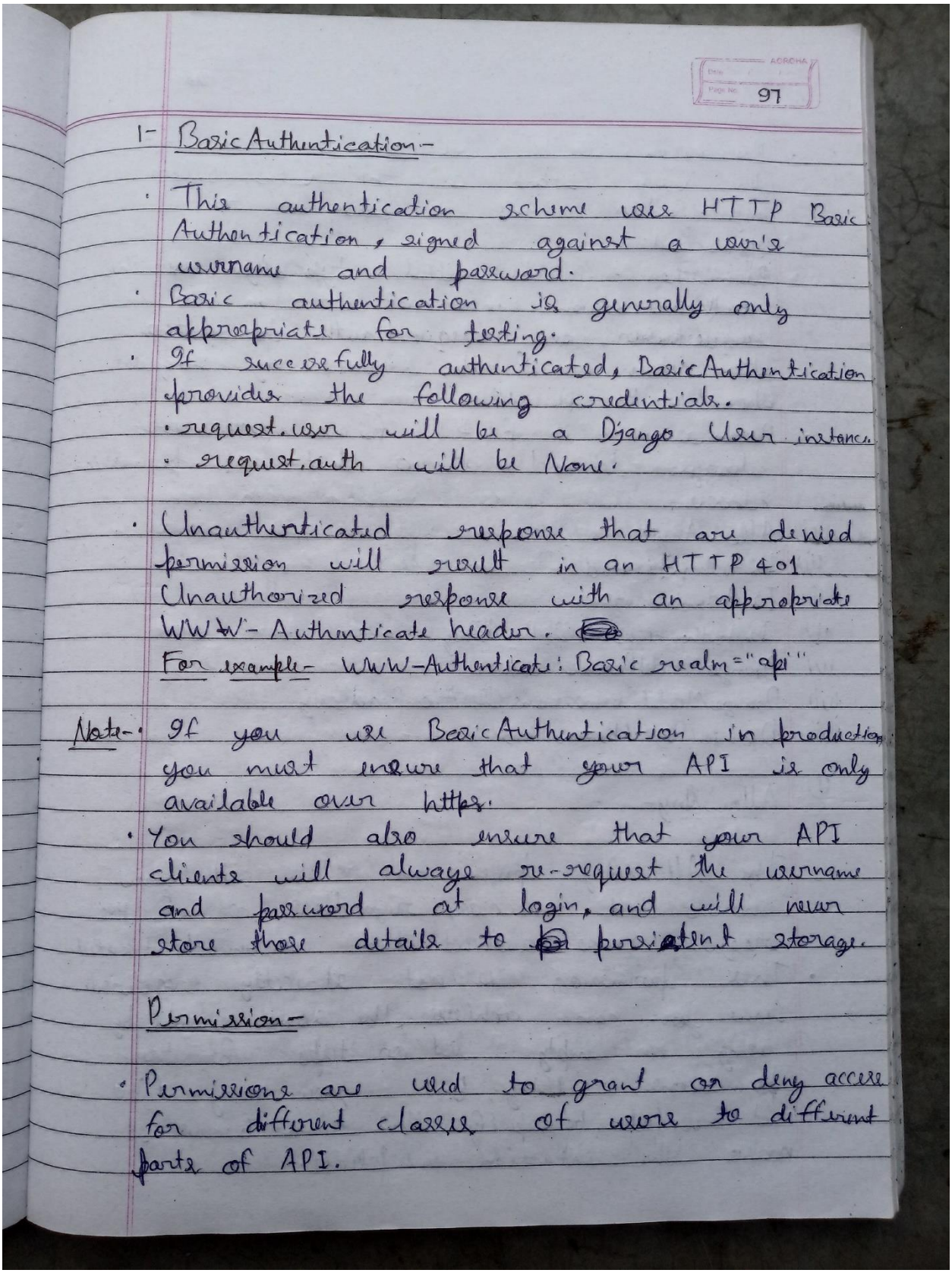
- Data is always associated with a creator
- Only authenticated users may create Data.
- Only the creator of a Data may update or delete it.
- Unauthenticated requests should have full read-only access.



Authentication-

- Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with. The permission and throttling policies can then use those credentials to determine if the request should be permitted.
- Authentication is always run at the very start of the view, before the permission and throttling run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.
- REST framework provides a number of authentication schemes out of the box, and also allows you to implement custom schemes.

1- Basic Authentication
2- Session Authication
3- Token Authentication
4- RemoteUserAuthentication
5- Custom authentication



1- Basic Authentication -

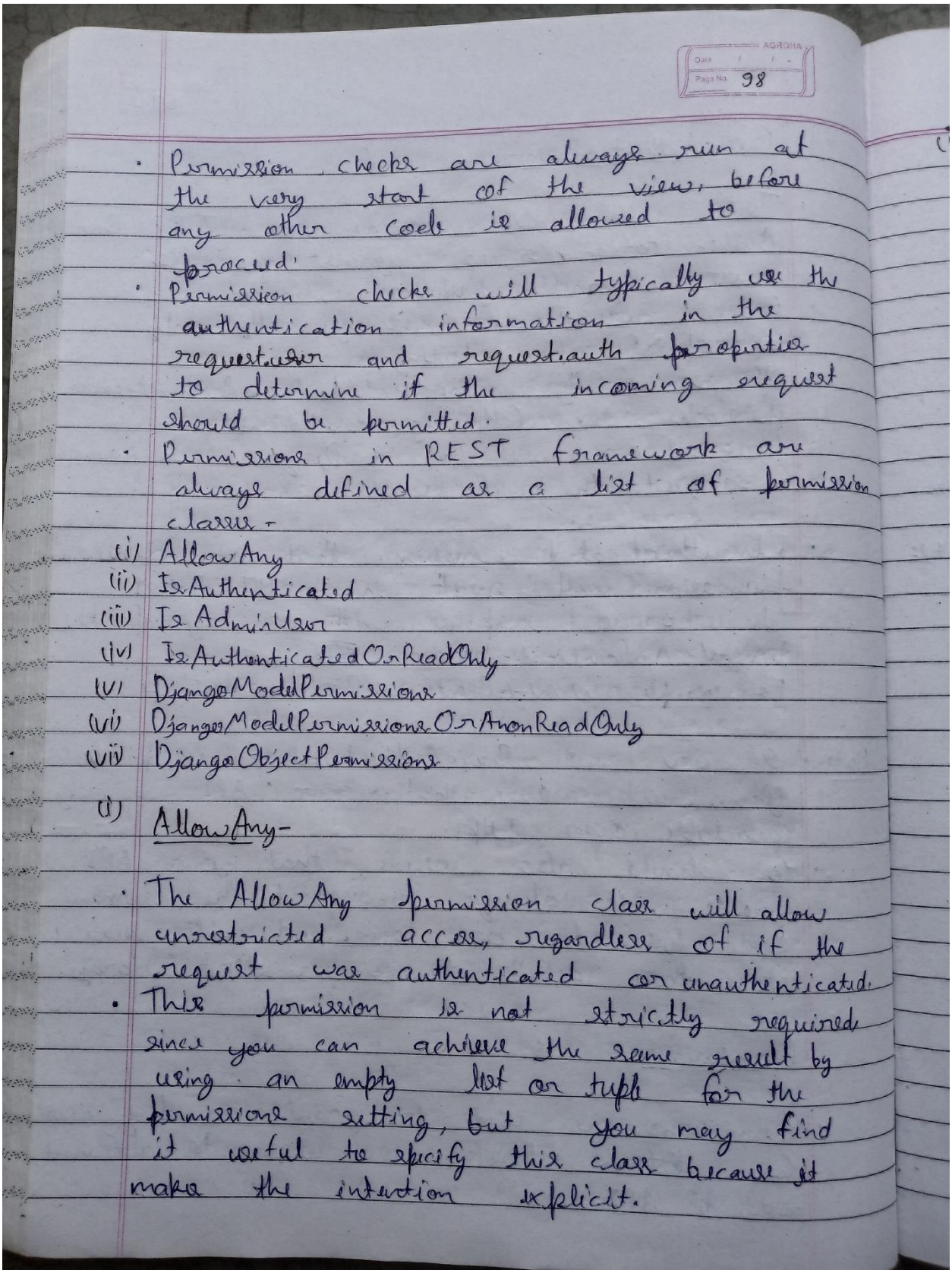
- This authentication scheme uses HTTP Basic Authentication, signed against a user's username and password.
- Basic authentication is generally only appropriate for testing.
- If successfully authenticated, BasicAuthentication provides the following credentials.
  - request.user will be a Django User instance.
  - request.auth will be None.
- Unauthenticated responses that are denied permission will result in an HTTP 401 Unauthorized response with an appropriate WWW-Authenticate header.

For example- WWW-Authenticate: Basic realm="api"

Note-
- If you use BasicAuthentication in production you must ensure that your API is only available over https.
- You should also ensure that your API clients will always re-request the username and password at login, and will never store those details to persistent storage.

Permission -

- Permissions are used to grant or deny access for different classes of users to different parts of API.

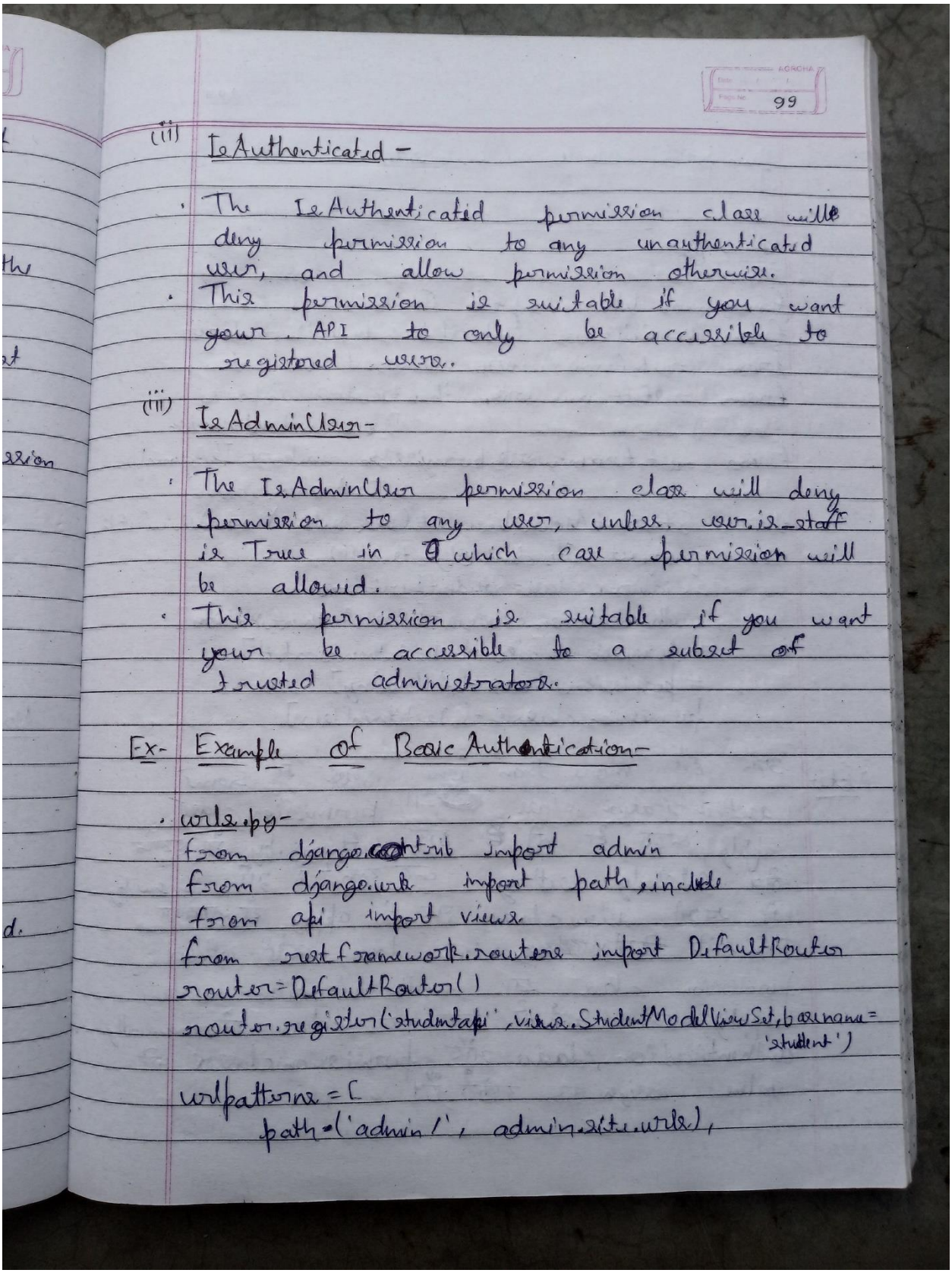


- Permission checks are always run at the very start of the view, before any other code is allowed to proceed.
- Permission checks will typically use the authentication information in the request.user and request.auth properties to determine if the incoming request should be permitted.
- Permissions in REST framework are always defined as a list of permission classes -

(i) AllowAny
(ii) IsAuthenticated
(iii) IsAdminUser
(iv) IsAuthenticatedOrReadOnly
(v) DjangoModelPermissions
(vi) DjangoModelPermissionsOrAnonReadOnly
(vii) DjangoObjectPermissions

(i) AllowAny -

- The AllowAny permission class will allow unrestricted access, regardless of if the request was authenticated or unauthenticated.
- This permission is not strictly required since you can achieve the same result by using an empty list or tuple for the permissions setting, but you may find it useful to specify this class because it makes the intention explicit.



(ii) IsAuthenticated -

- The IsAuthenticated permission class will deny permission to any unauthenticated user, and allow permission otherwise.
- This permission is suitable if you want your API to only be accessible to registered users.

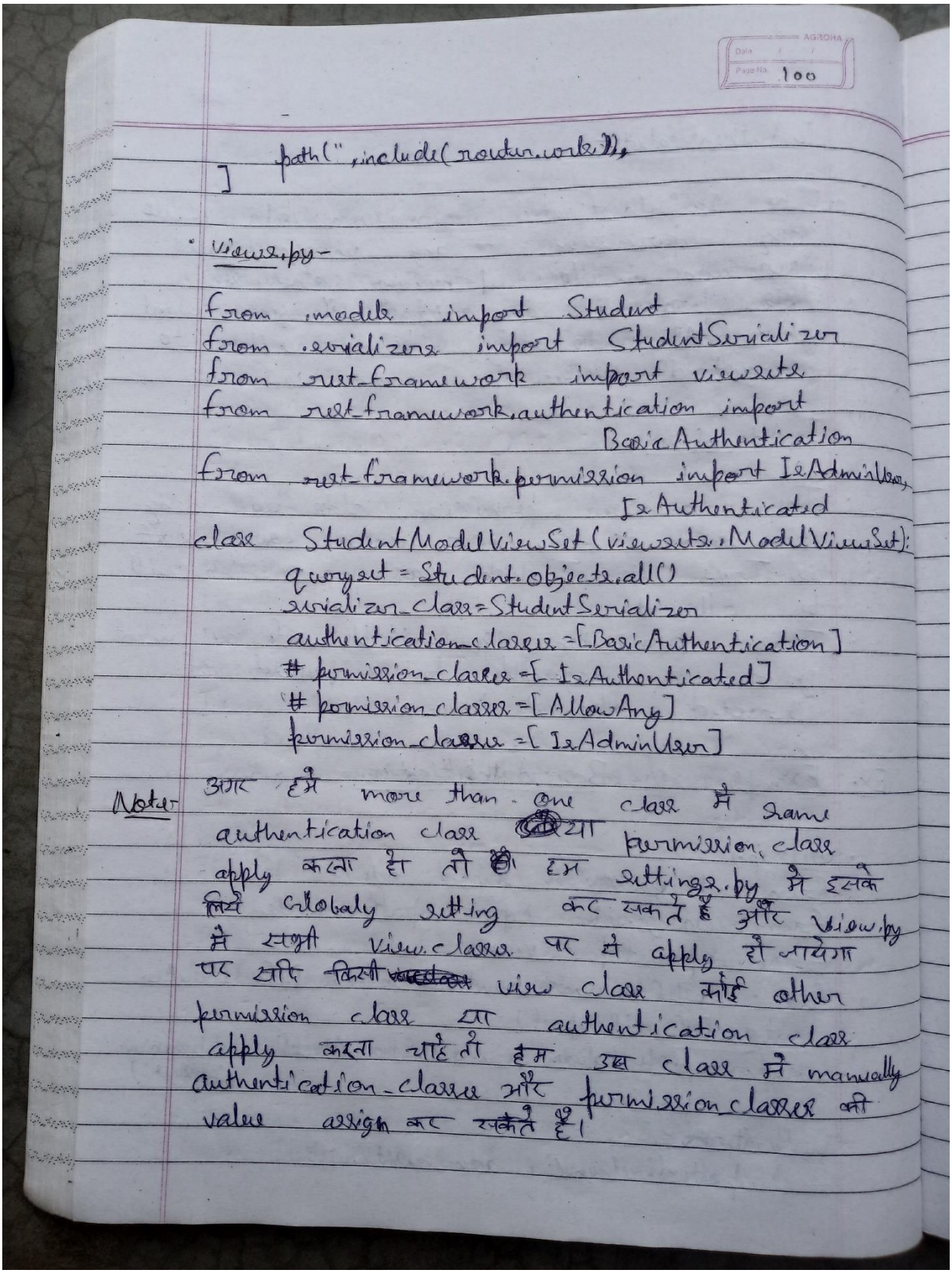
(iii) IsAdminUser -

- The IsAdminUser permission class will deny permission to any user, unless user.is_staff is True in which case permission will be allowed.
- This permission is suitable if you want your be accessible to a subset of trusted administrators.

Ex- Example of Basic Authentication -

- urls.py -

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register('studentapi', views.StudentModelViewSet, basename='student')

urlpatterns = [
    path('admin/', admin.site.urls),
```

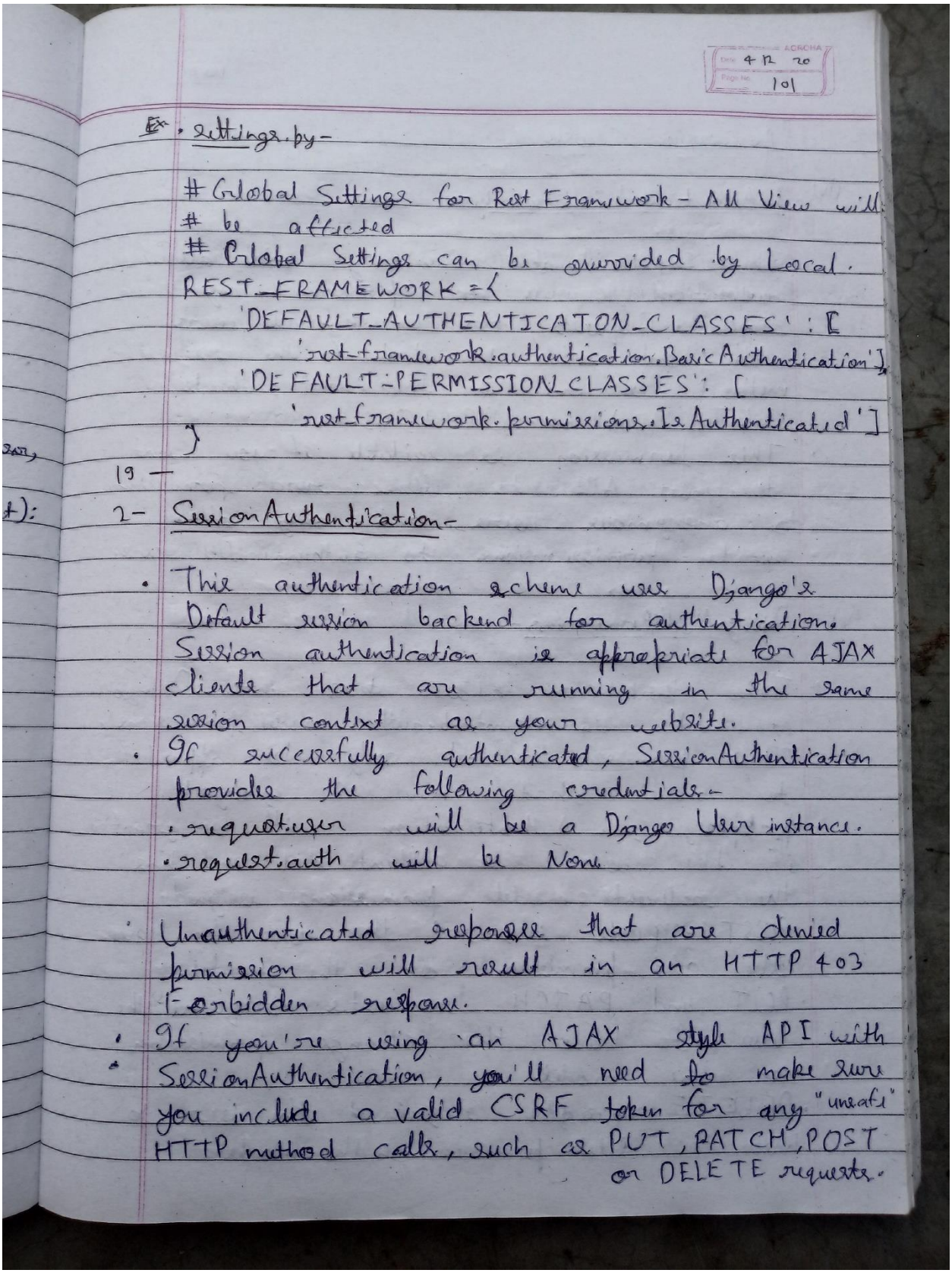


path('', include(router.urls)),
]

- views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
from rest_framework.authentication import
                                BasicAuthentication
from rest_framework.permission import IsAdminUser,
                                IsAuthenticated
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [BasicAuthentication]
    # permission_classes = [IsAuthenticated]
    # permission_classes = [AllowAny]
    permission_classes = [IsAdminUser]
```

Note:- अगर हमें more than one class में same authentication class या permission class apply करना है तो हम settings.py में इसके लिये Globaly setting कर सकते हैं और views.py में सभी view classes पर ये apply हो जायेगा पर यदि किसी view class में other permission class या authentication class apply करना चाहें तो हम उस class में manually authentication_classes और permission_classes की value assign कर सकते हैं।



Ex. • settings.py -

```
# Global Settings for Rest Framework - All Views will
# be affected
# Global Settings can be overrided by Local.
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATON_CLASSES' : [
        'rest_framework.authentication.BasicAuthentication'],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated']
}
```

19 —

2- SessionAuthentication -

- This authentication scheme uses Django's Default session backend for authentication. Session authentication is appropriate for AJAX clients that are running in the same session context as your website.
- If successfully authenticated, SessionAuthentication provides the following credentials -
  - request.user will be a Django User instance.
  - request.auth will be None.
- Unauthenticated responses that are denied permission will result in an HTTP 403 Forbidden response.
- If you're using an AJAX style API with SessionAuthentication, you'll need to make sure you include a valid CSRF token for any "unsafe" HTTP method calls, such as PUT, PATCH, POST or DELETE requests.

Permission Classes -

(iv) IsAuthenticatedOrReadOnly -

- The IsAuthenticatedOrReadOnly will allow authenticated users to perform any request. Requests for unauthorised users will only be permitted if the request method is one of the "safe" methods; GET, HEAD or OPTIONS.
- This permission is suitable if you want to your API to allow read permissions to anonymous users, and only allow write permissionsions to authenticated users.

(v) DjangoModelPermissions -

- This permission class ties into Django's standard django.contrib.auth model permissions. This permission must only be applied to views that have a queryset property set. Authorization will be granted if the user is authenticated and has the relevent model permissions assigned.
- POST requests required the user to have the add permission on the model.
- PUT and PATCH requests require the user to have the change permission on the model.
- DELETE requests required the user to have the delete permission on the model.

- The default behaviour can also be overridden to support custom model permissions. For example, you might want to include a view model permission for GET requests.
- To use custom model permissions, override DjangoModelPermissions and set the perms_map property.

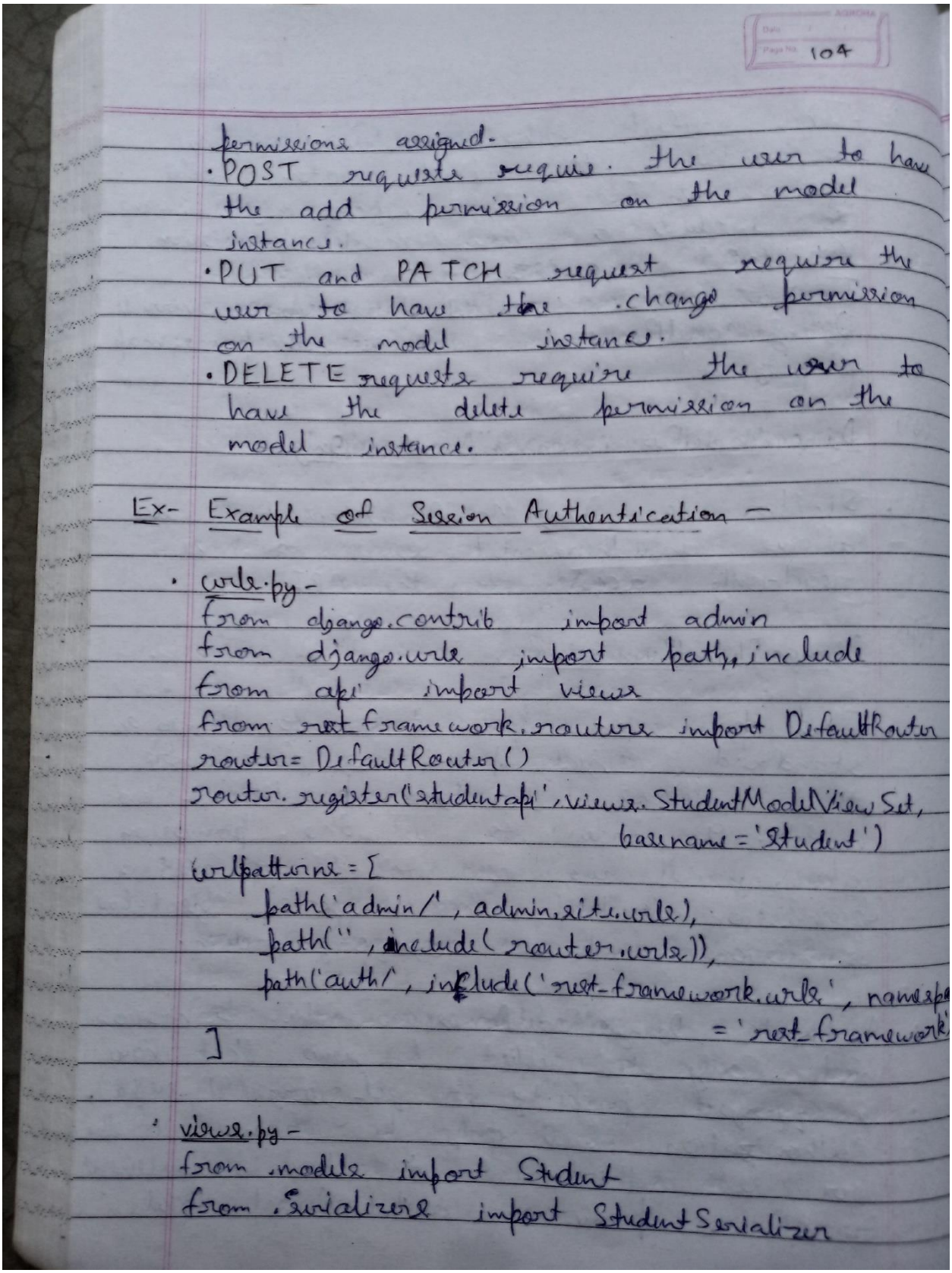
(vi) DjangoModelPermissionsOrAnonReadOnly -

- Similar to DjangoModelPermissions, but also allows unauthenticated users to have read-only access to the API.

(vii) DjangoObjectPermissions -

- This permission class ties into Django's standard object permissions framework that allows per-object permissions on models. In order to use this permission class, you'll also need to add ~~to~~ ~~add~~ a permission backend that supports object-level permissions, such as django-guardian.

- As with DjangoModelPermissions, this permission must only be applied to views that have a queryset property or get_queryset() method. Authorization will only be granted if the user is authenticated and has the relevant per-object permissions and relevant model



permissions assigned.
- POST requests require the user to have the add permission on the model instance.
- PUT and PATCH request require the user to have the change permission on the model instance.
- DELETE requests require the user to have the delete permission on the model instance.

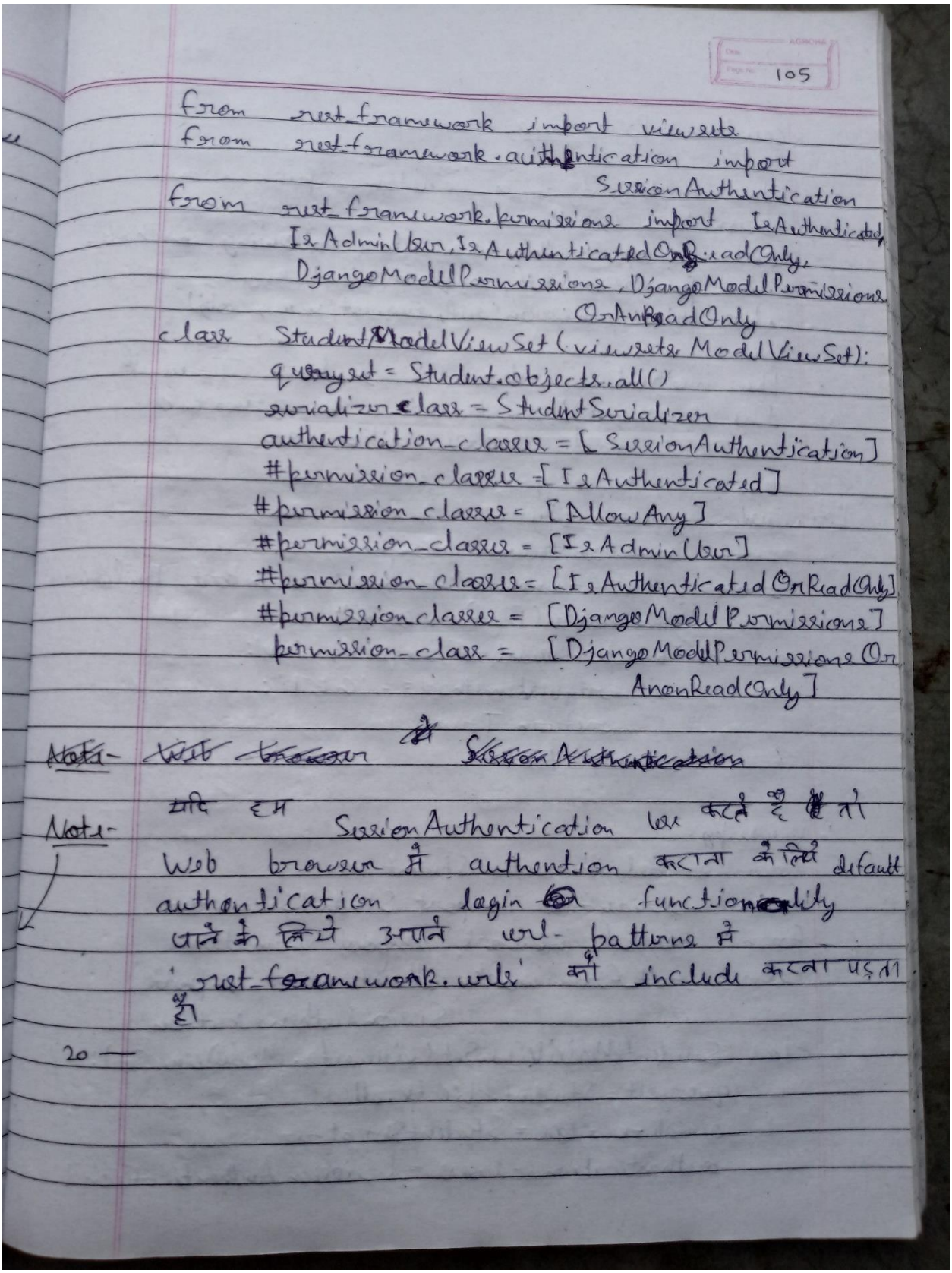
Ex- Example of Session Authentication –

- urls.py –

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register('studentapi', views.StudentModelViewSet,
                                basename = 'Student')
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
    path('auth/', include('rest_framework.urls', namespa
                                = 'rest_framework')
]
```

- views.py –

```
from .models import Student
from .serializers import StudentSerializer
```



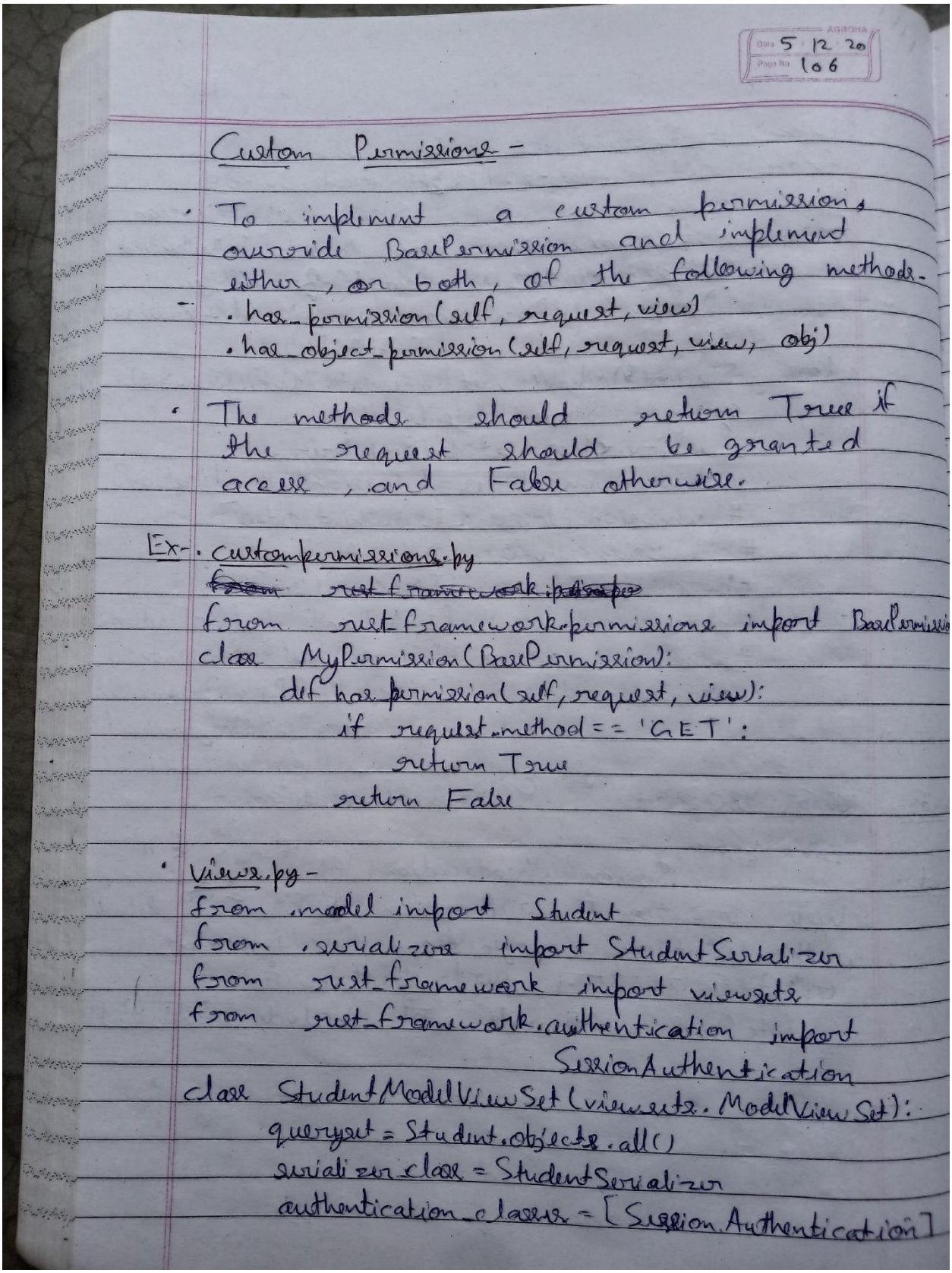
105

```
from rest_framework import viewsets
from rest_framework.authentication import
                                SessionAuthentication
from rest_framework.permissions import IsAuthenticated,
    IsAdminUser, IsAuthenticatedOrReadOnly,
    DjangoModelPermissions, DjangoModelPermissions
                                OrAnonReadOnly
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [SessionAuthentication]
    #permission_classes = [IsAuthenticated]
    #permission_classes = [AllowAny]
    #permission_classes = [IsAdminUser]
    #permission_classes = [IsAuthenticatedOrReadOnly]
    #permission_classes = [DjangoModelPermissions]
    permission_class = [DjangoModelPermissions Or
                                AnonReadOnly]
```

Note- ~~Web browser~~ ~~SessionAuthentication~~

Note- यदि हम SessionAuthentication use करते है तो Web browser में authention कराना के लिये default authentication login functionality पाने के लिये उपने url-patterns में 'rest_framework.urls' को include करना पड़ता है।

20 —



Custom Permissions -

- To implement a custom permissions, override BasePermission and implement either, or both, of the following methods.
  - has_permission(self, request, view)
  - has_object_permission(self, request, view, obj)

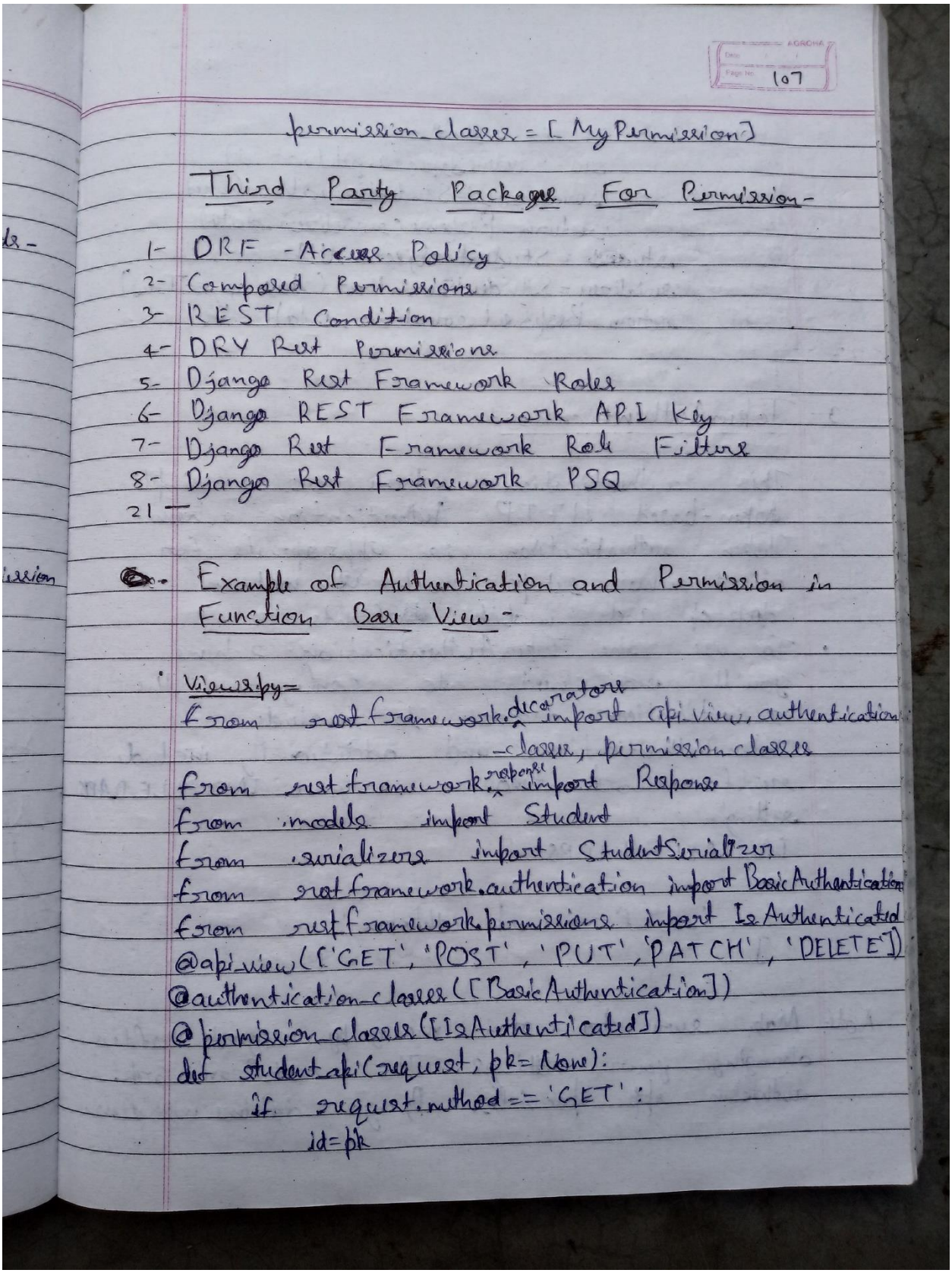
- The methods should return True if the request should be granted access, and False otherwise.

Ex- • custompermissions.py

```
from rest_framework.permissions import BasePermission
class MyPermission(BasePermission):
    def has_permission(self, request, view):
        if request.method == 'GET':
            return True
        return False
```

- views.py -

```
from .model import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
from rest_framework.authentication import
                                SessionAuthentication
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [SessionAuthentication]
```



permission_classes = [MyPermission]

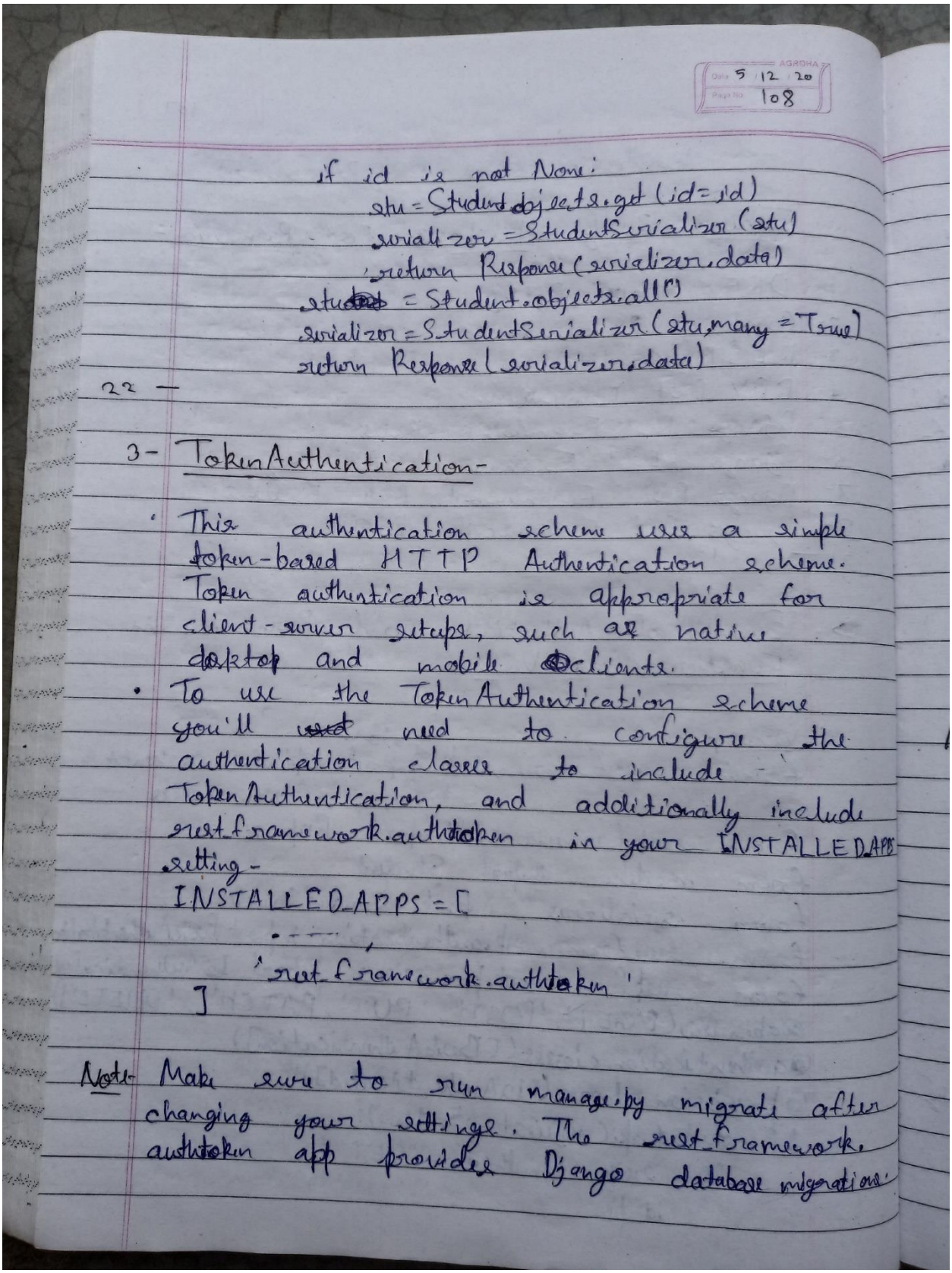
Third Party Package For Permission-

1- DRF - Access Policy
2- Composed Permissions
3- REST Condition
4- DRY Rest Permissions
5- Django Rest Framework Roles
6- Django REST Framework API Key
7- Django Rest Framework Roli Filters
8- Django Rest Framework PSQ
21 —

Example of Authentication and Permission in Function Base View -

views.py=

```
from rest_framework.decorators import api_view, authentication_classes, permission_classes
from rest_framework.response import Response
from .models import Student
from .serializers import StudentSerializer
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated
@api_view(['GET', 'POST', 'PUT', 'PATCH', 'DELETE'])
@authentication_classes([BasicAuthentication])
@permission_classes([IsAuthenticated])
def student_api(request, pk=None):
    if request.method == 'GET':
        id=pk
```



```
        if id is not None:
            stu = Student.objects.get(id=id)
            serializer = StudentSerializer(stu)
            return Response(serializer.data)
        stu = Student.objects.all()
        serializer = StudentSerializer(stu, many=True)
        return Response(serializer.data)
```

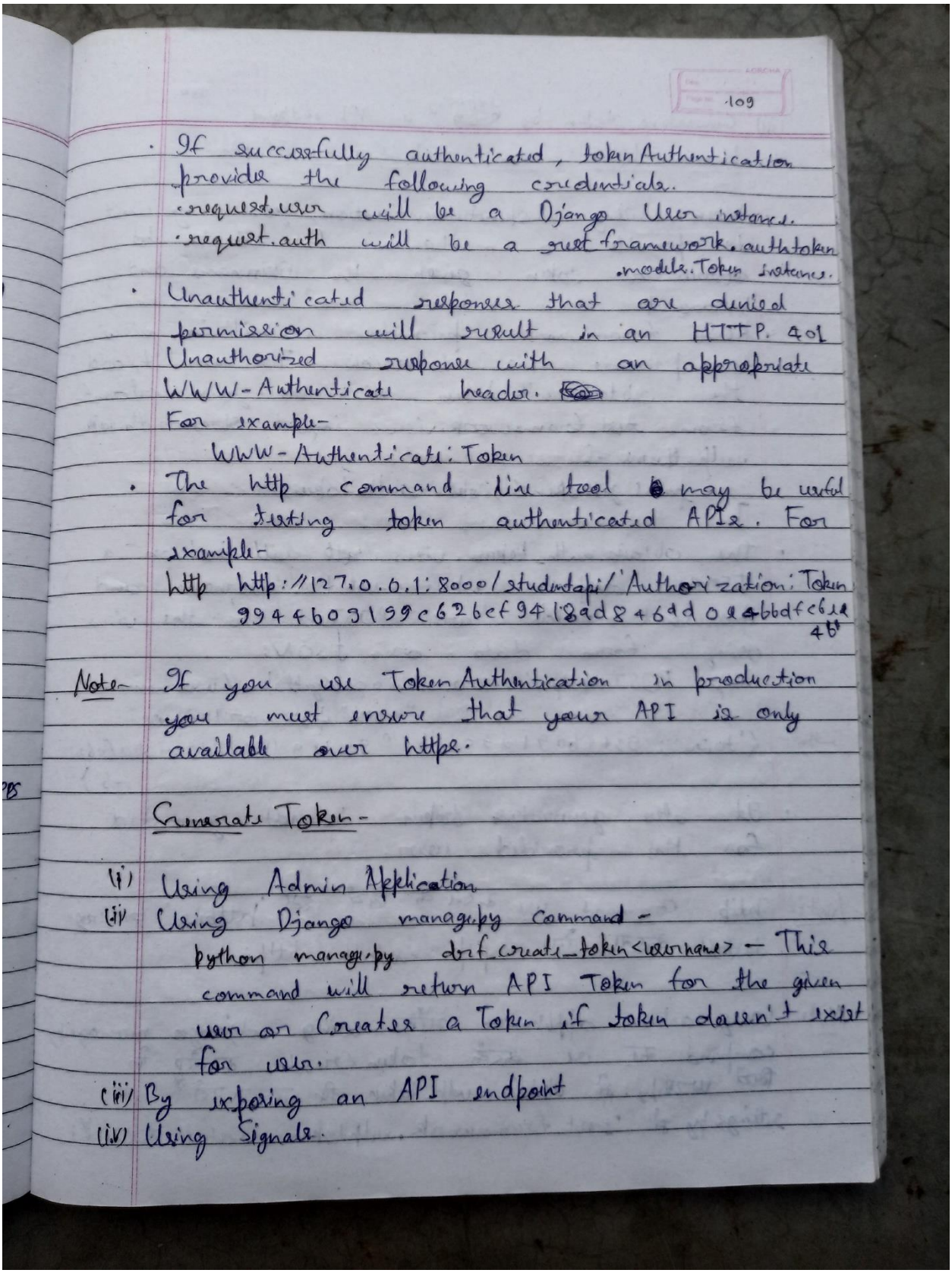
22 —

3- TokenAuthentication-

- This authentication scheme uses a simple token-based HTTP Authentication scheme. Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.
- To use the TokenAuthentication scheme you'll need to configure the authentication classes to include TokenAuthentication, and additionally include rest_framework.authtoken in your INSTALLED_APPS setting-

```
INSTALLED_APPS = [
    ....,
    'rest_framework.authtoken'
]
```

Note- Make sure to run manage.py migrate after changing your settings. The rest_framework.authtoken app provides Django database migrations.



- If successfully authenticated, TokenAuthentication provides the following credentials.
  - request.user will be a Django User instance.
  - request.auth will be a rest_framework.authtoken.models.Token instance.
- Unauthenticated responses that are denied permission will result in an HTTP 401 Unauthorized response with an appropriate WWW-Authenticate header.
  For example-
  WWW-Authenticate: Token
- The http command line tool may be useful for testing token authenticated APIs. For example-
  http http://127.0.0.1:8000/studentapi/ 'Authorization: Token 9944b09199c62bcf9418ad846d0e4bbdfc6ee4b'

Note- If you use TokenAuthentication in production you must ensure that your API is only available over https.

Generate Token-

(i) Using Admin Application
(ii) Using Django manage.py command -
python manage.py drf_create_token <username> - This command will return API Token for the given user or Creates a Token if token doesn't exist for user.
(iii) By exposing an API endpoint
(iv) Using Signals.

(iii) Generating token by ~~deleting~~ using an API endpoint -

How Client can Ask/Create Token-

- When using TokenAuthentication, you may want to provide a machanism for clients to obtain a token given the username and password.
- REST framework provides a built-in view to provide this behaviour. To use it, add the obtain_auth_token view to your urlconf.

```
from rest_framework.views import obtain_auth_token
urlpatterns = [
    path('gettoken/', obtain_auth_token)
]
```

- The obtain_auth_token view will return a JSON response when valid username and password fields are POSTed to the view using form data or JSON:-

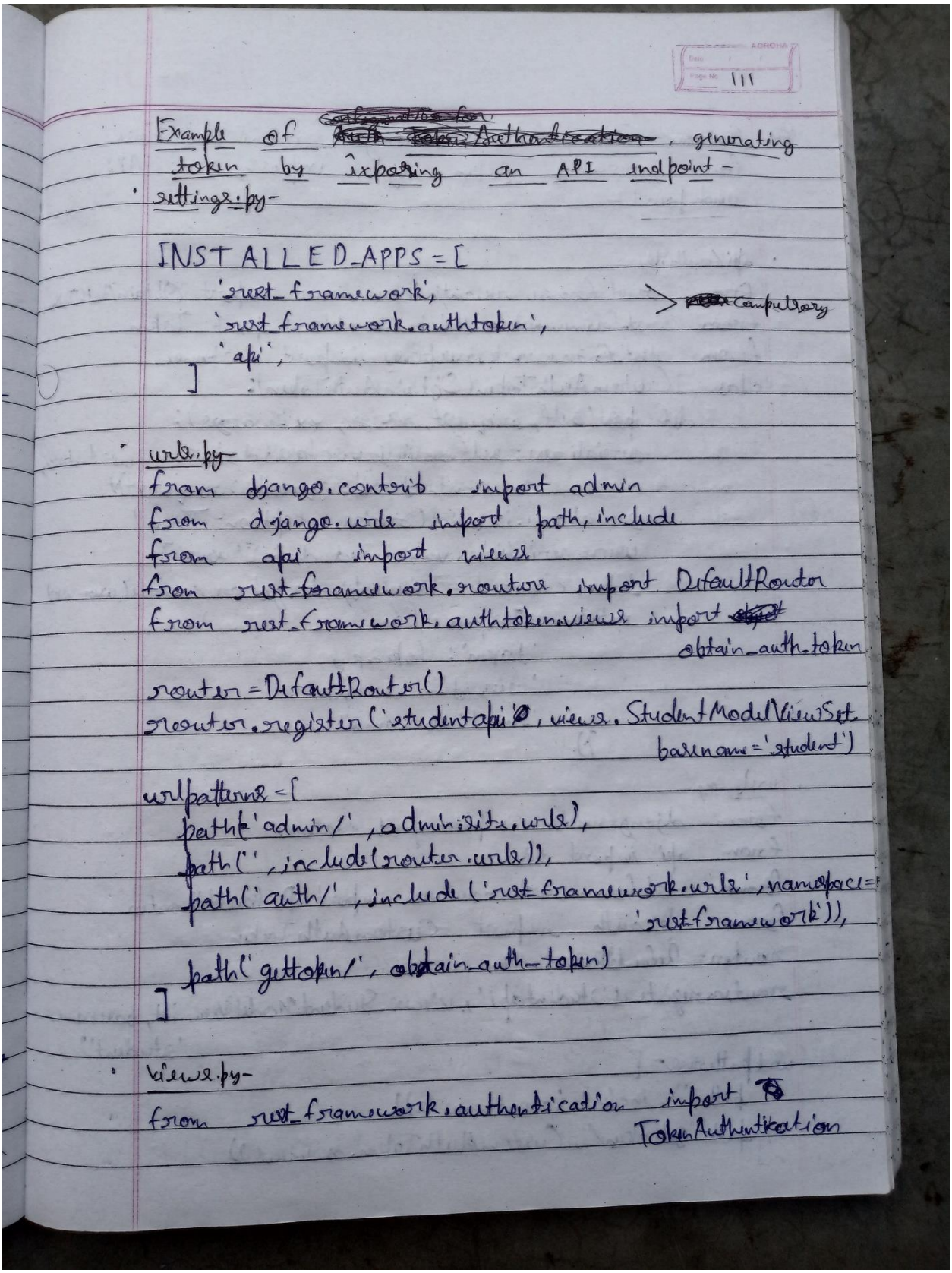
```
http POST http://127.0.0.1:8000/gettoken/ username="name" password="pass"
```

output- {'token': '99+4b09199c62bcf9418ad846dd0e4bbdfc6ee4b'}

- It also generates token is not generated for the provided user.

Note- http command use करने के लिये हमें 'httpie' package install करना होगा - pip install httpie

Note- using admin application या फिर using django management command का use करके token create करवाने के लिये views.py में obtain_auth_token को import करना नहीं है बस settings.py में 'rest_framework.authtoken' install करना है।



Example of ~~Token Authentication~~, generating token by ixposing an API endpoint -

- settings.py -

```
INSTALLED_APPS = [
    'rest_framework',
    'rest_framework.authtoken',
    'api',
]
```

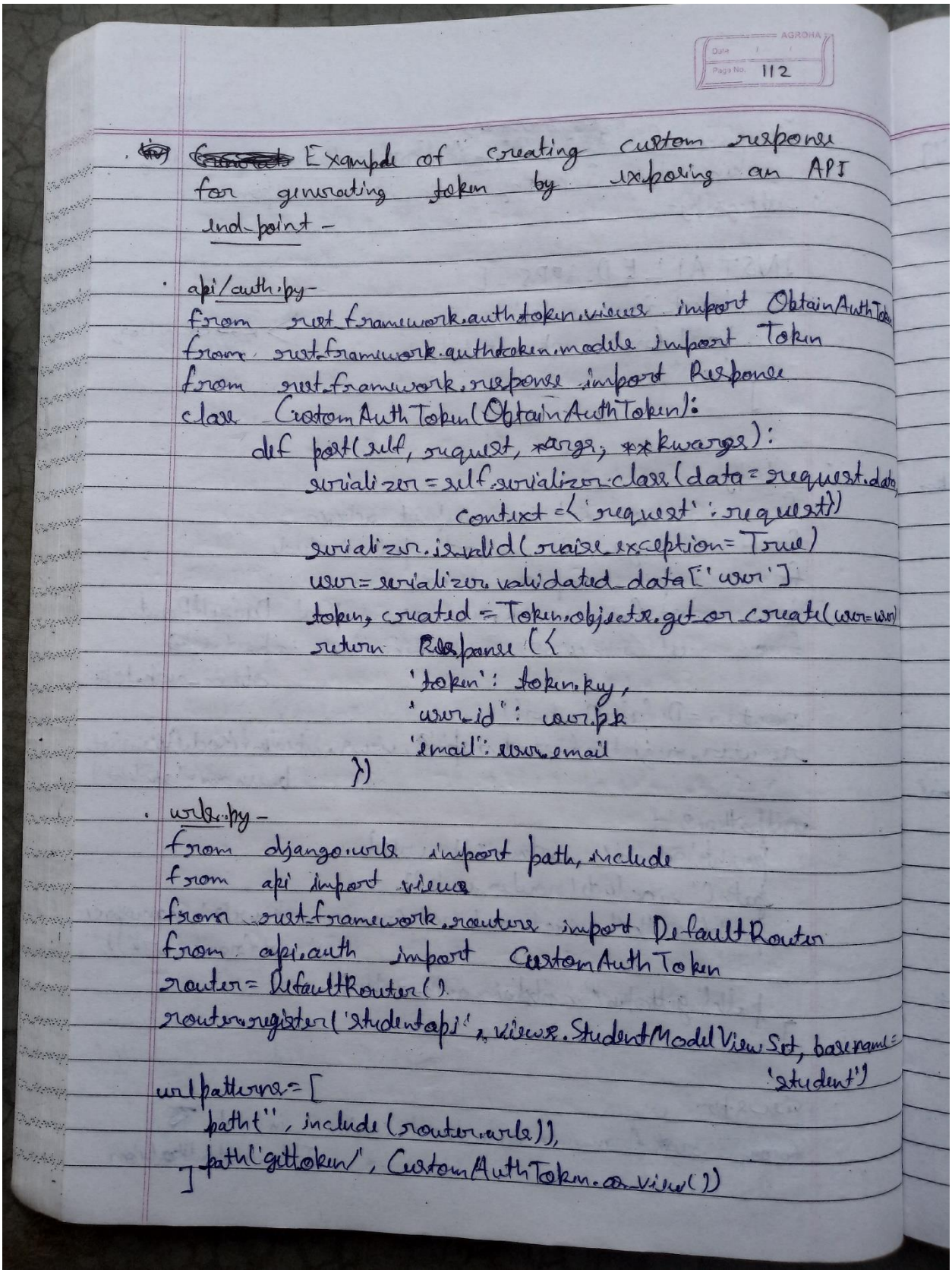
> ~~Not~~ Compulsory

- urls.py

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
from rest_framework.authtoken.views import obtain_auth_token

router = DefaultRouter()
router.register('studentapi', views.StudentModelViewSet,
                basename='student')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
    path('auth/', include('rest_framework.urls', namespace=
                          'rest_framework')),

    path('gettoken/', obtain_auth_token)
]
```

- views.py -

```
from rest_framework.authentication import
                    TokenAuthentication
```



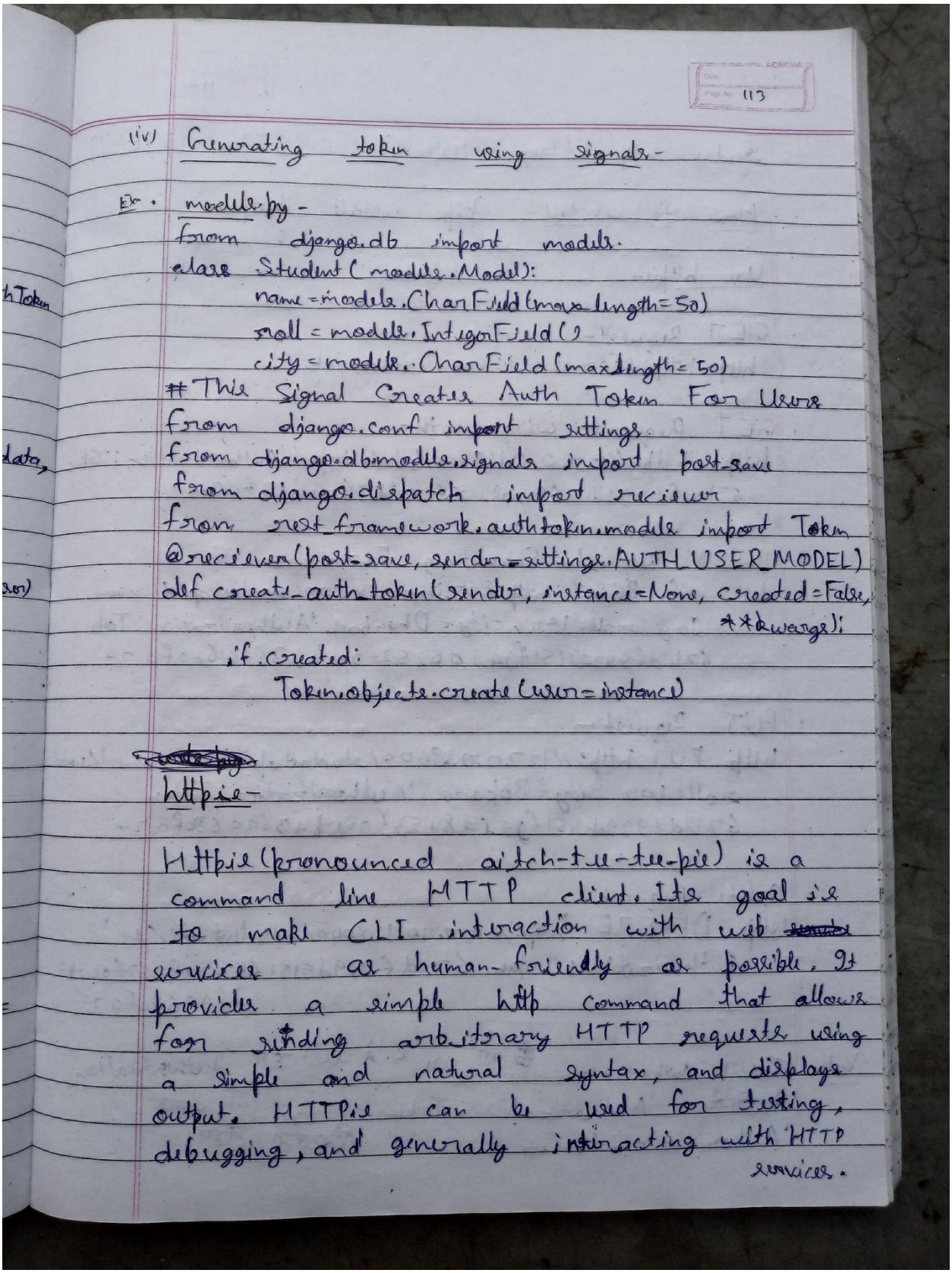
Example of creating custom response for generating token by exposing an API end-point -

- api/auth.py -

```
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.authtoken.models import Token
from rest_framework.response import Response
class CustomAuthToken(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data,
                                           context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'user_id': user.pk
            'email': user.email
        })
```

- urls.py -

```
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
from api.auth import CustomAuthToken
router = DefaultRouter()
router.register('studentapi', views.StudentModelViewSet, basename=
                                                    'student')
urlpatterns = [
    path('', include(router.urls)),
    path('gettoken/', CustomAuthToken.as_view())
]
```



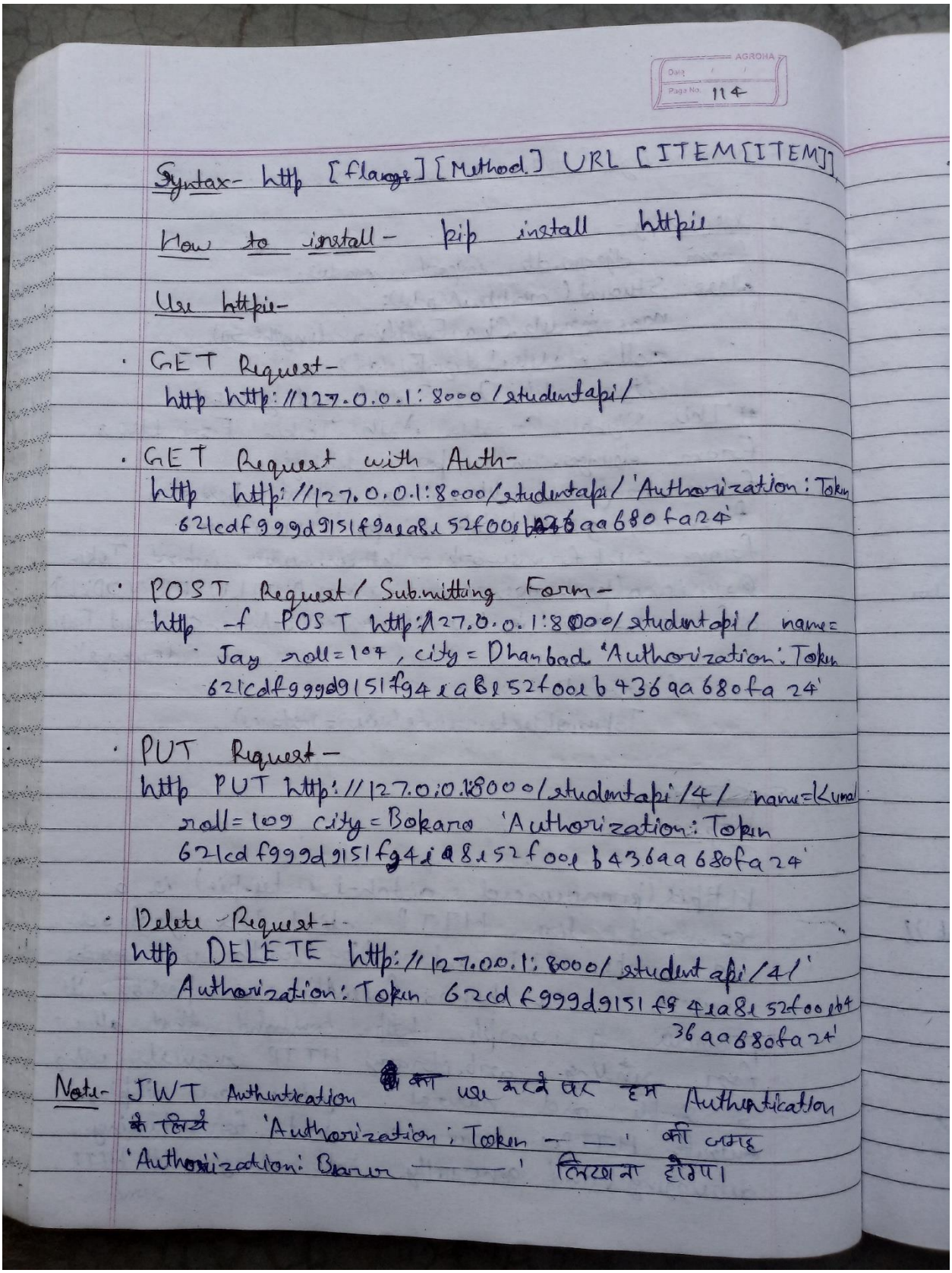
(iv) Generating token using signals -

Ex. models.py -

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.IntegerField()
    city = models.CharField(max_length=50)
# This Signal Creates Auth Token For Users
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token
@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False,
                      **kwargs):
    if created:
        Token.objects.create(user=instance)
```

httpie -

Httpie (pronounced aitch-tee-tee-pie) is a command line HTTP client. Its goal is to make CLI interaction with web services as human-friendly as possible. It provides a simple http command that allows for sending arbitrary HTTP requests using a simple and natural syntax, and displays output. HTTPie can be used for testing, debugging, and generally interacting with HTTP services.



Syntax- http [flags] [Method] URL [ITEM[ITEM]]

How to install- pip install httpie

Use httpie-

- GET Request-
  http http://127.0.0.1:8000/studentapi/

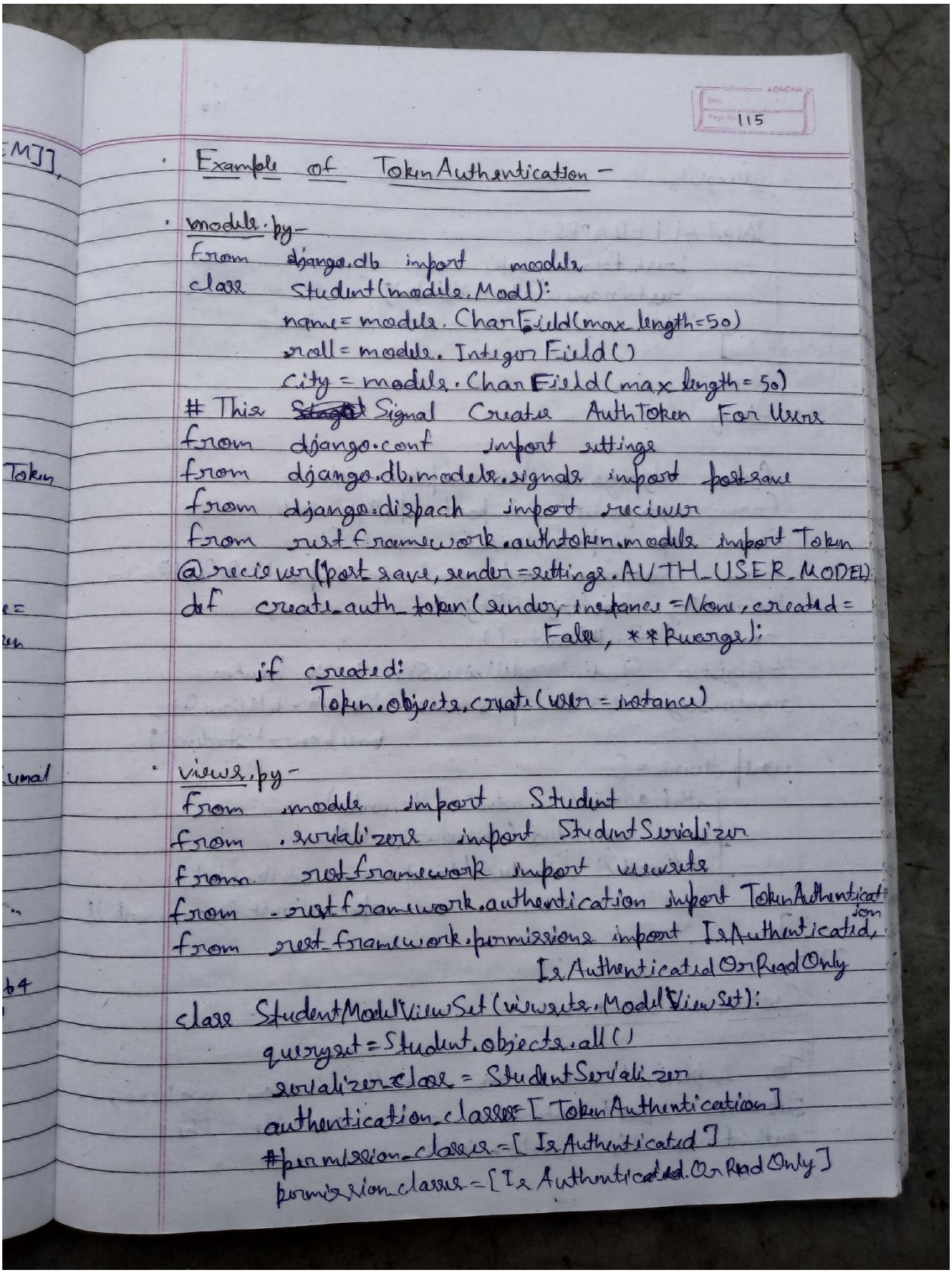
- GET Request with Auth-
  http http://127.0.0.1:8000/studentapi/ 'Authorization: Token 621cdf999d9151f94a8a52f001b436aa680fa24'

- POST Request / Submitting Form-
  http -f POST http://127.0.0.1:8000/studentapi/ name=Jay roll=104, city=Dhanbad 'Authorization: Token 621cdf999d9151f94a8a52f001b436aa680fa24'

- PUT Request-
  http PUT http://127.0.0.1:8000/studentapi/4/ name=Kunal roll=109 city=Bokaro 'Authorization: Token 621cdf999d9151f94a8a52f001b436aa680fa24'

- Delete Request-
  http DELETE http://127.0.0.1:8000/studentapi/4/ 'Authorization: Token 621cdf999d9151f94a8a52f001b436aa680fa24'

Note- JWT Authentication का use करने पर हम Authentication के लिये 'Authorization: Token ——' की जगह 'Authorization: Bearer ——' लिखना होगा।



Page No. 115

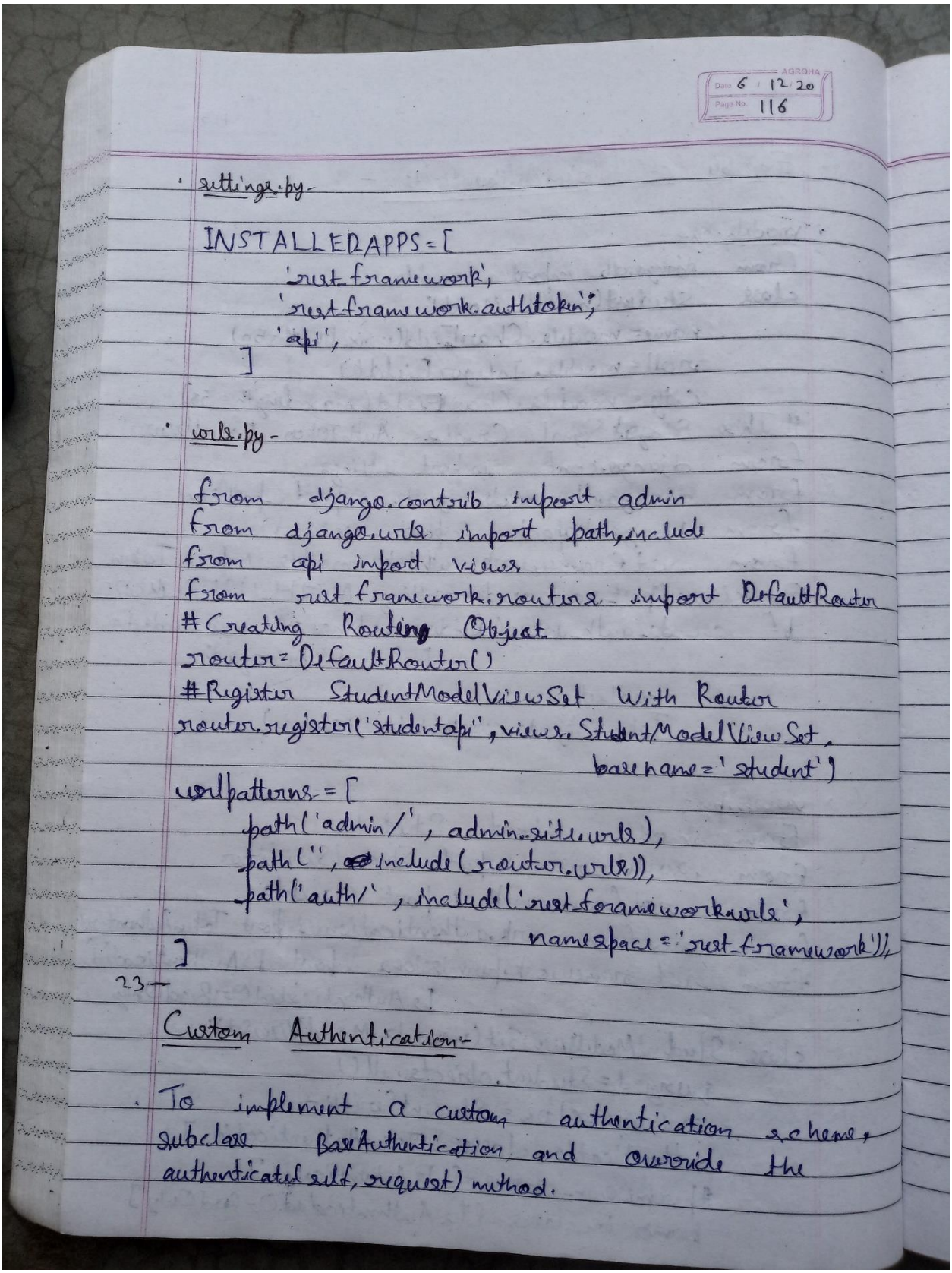
- Example of TokenAuthentication -

- models.py -

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.IntegerField()
    city = models.CharField(max_length=50)
# This Signal Create AuthToken For Users
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token
@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=
                      False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

- views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
from .rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated,
                                       IsAuthenticatedOrReadOnly
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [TokenAuthentication]
    #permission_classes = [IsAuthenticated]
    permission_classes = [IsAuthenticatedOrReadOnly]
```



- settings.py -

```
INSTALLED_APPS = [
    'rest_framework',
    'rest_framework.authtoken',
    'api',
]
```

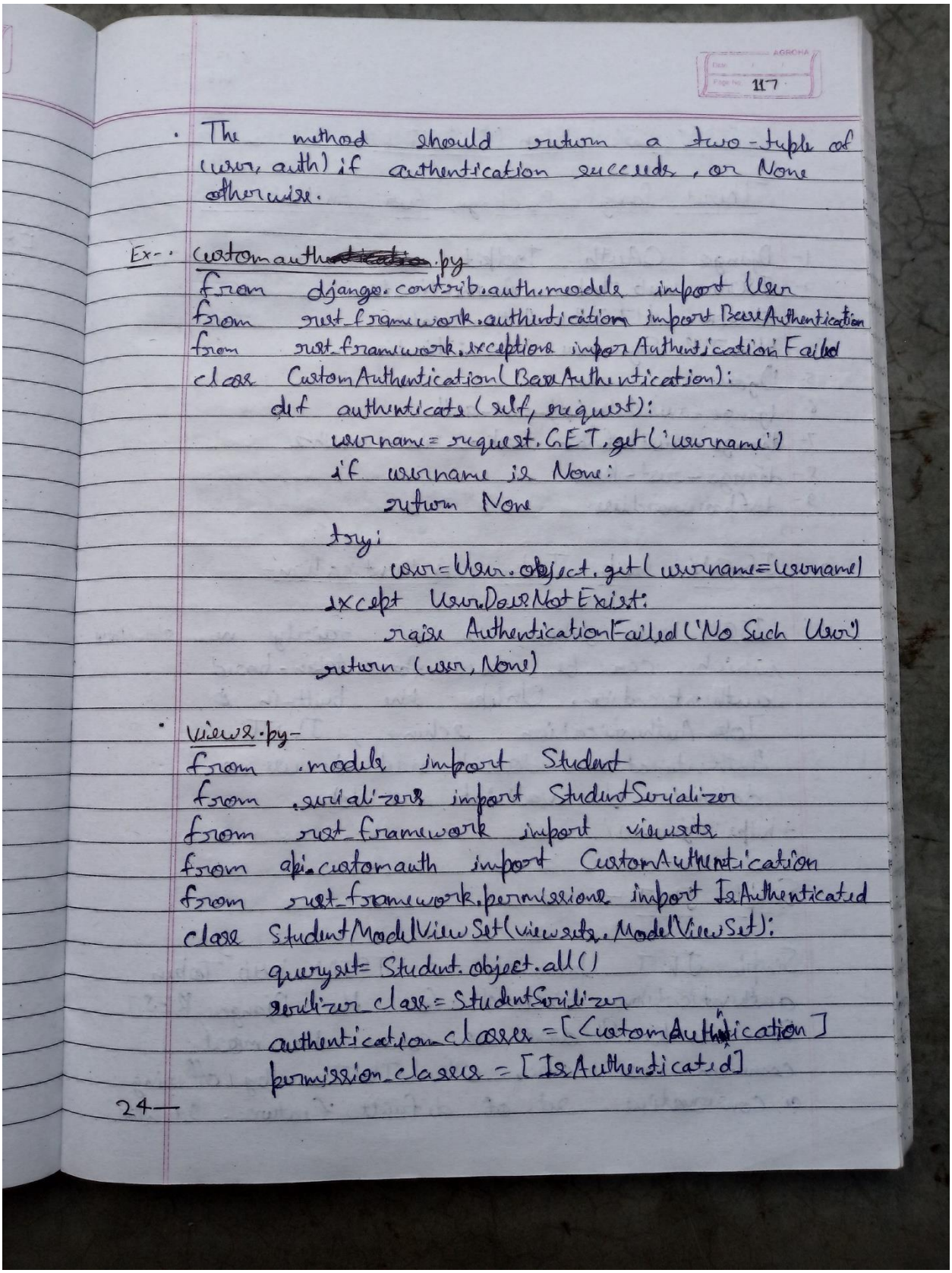
- urls.py -

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
#Creating Routing Object
router = DefaultRouter()
#Register StudentModelViewSet With Router
router.register('studentapi', views.StudentModelViewSet,
                basename='student')
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
    path('auth/', include('rest_framework.urls',
                namespace='rest_framework')),
]
```

23 —

Custom Authentication -

- To implement a custom authentication scheme, subclass BaseAuthentication and override the authenticate(self, request) method.



- The method should return a two-tuple of (user, auth) if authentication succeeds, or None otherwise.

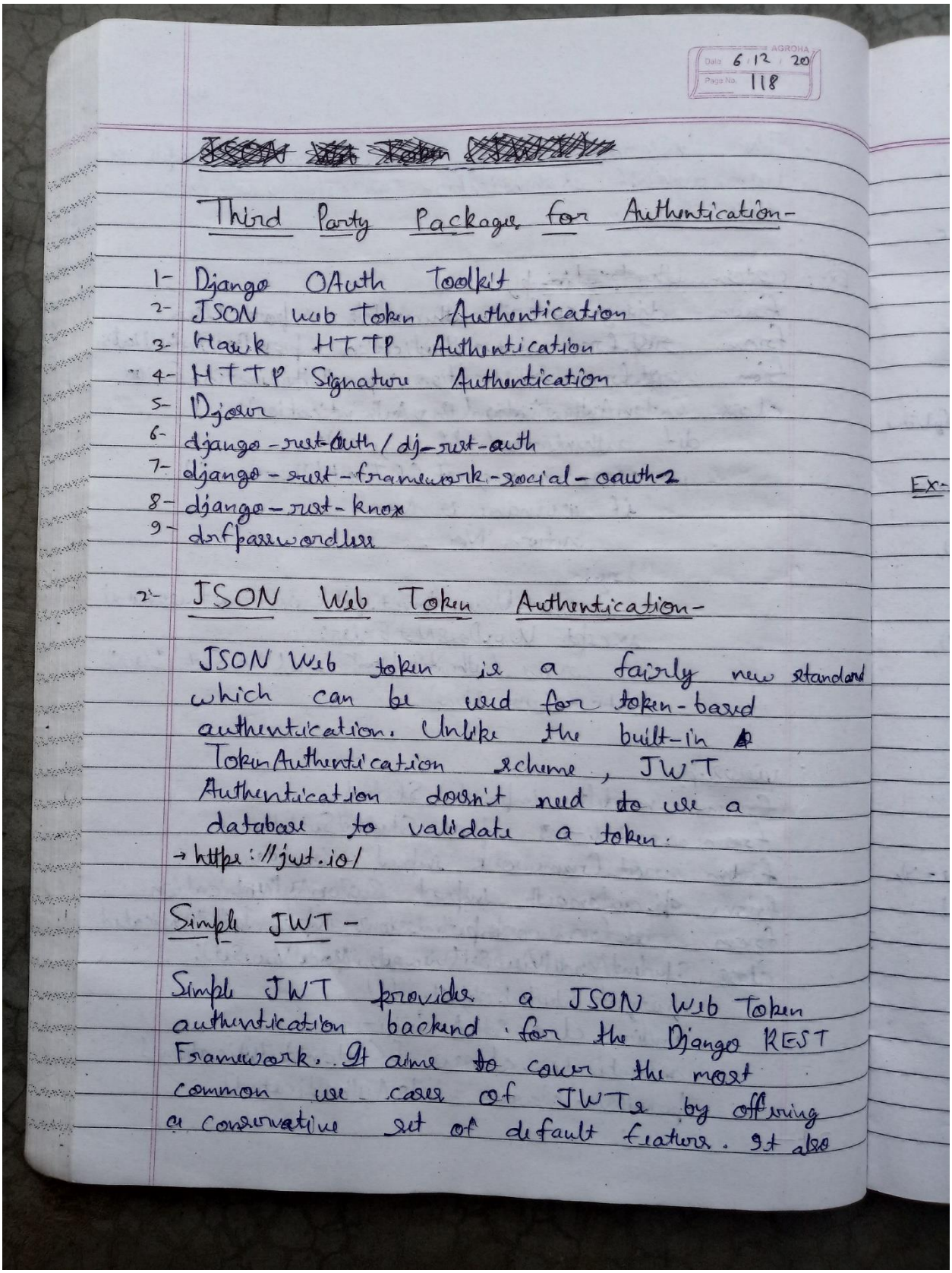
Ex-- customauth.py

```
from django.contrib.auth.models import User
from rest_framework.authentication import BaseAuthentication
from rest_framework.exceptions import AuthenticationFailed
class CustomAuthentication(BaseAuthentication):
    def authenticate(self, request):
        username = request.GET.get('username')
        if username is None:
            return None
        try:
            user = User.object.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('No Such User')
        return (user, None)
```

- views.py-

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
from api.customauth import CustomAuthentication
from rest_framework.permissions import IsAuthenticated
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.object.all()
    serializer_class = StudentSerializer
    authentication_classes = [CustomAuthentication]
    permission_classes = [IsAuthenticated]
```

24—



Third Party Package for Authentication-

1- Django OAuth Toolkit
2- JSON web Token Authentication
3- Hawk HTTP Authentication
4- HTTP Signature Authentication
5- Djoser
6- django-rest-auth / dj-rest-auth
7- django-rest-framework-social-oauth2
8- django-rest-knox
9- drfpasswordless

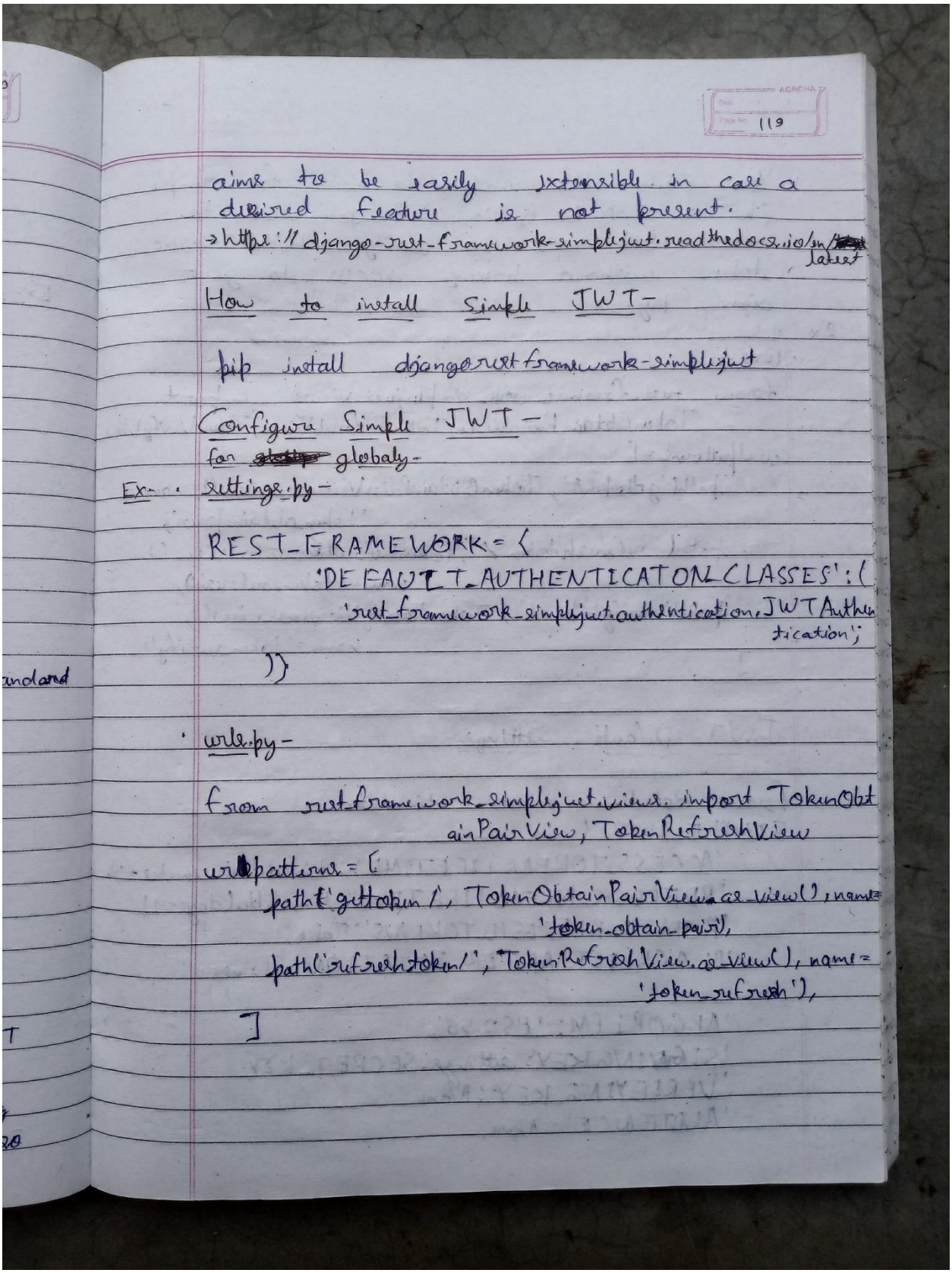
2- JSON Web Token Authentication-

JSON Web token is a fairly new standard which can be used for token-based authentication. Unlike the built-in TokenAuthentication scheme, JWT Authentication doesn't need to use a database to validate a token.
→ https://jwt.io/

Simple JWT -

Simple JWT provides a JSON Web Token authentication backend for the Django REST Framework. It aims to cover the most common use cases of JWTs by offering a conservative set of default features. It also

Ex-



aims to be easily extensible in case a desired feature is not present.
→ https://django-rest-framework-simplejwt.readthedocs.io/en/latest

How to install Simple JWT-

pip install djangorestframework-simplejwt

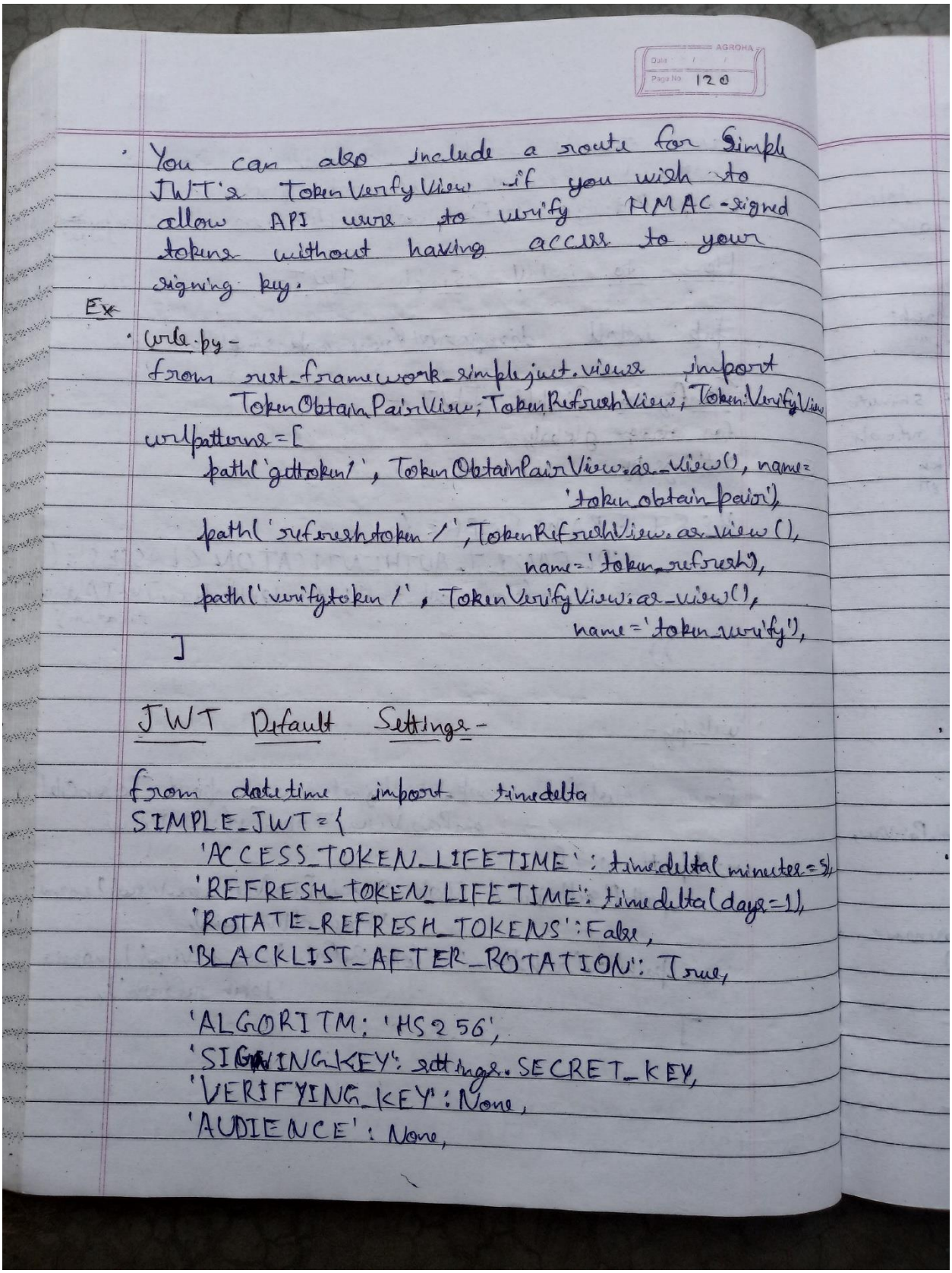
Configure Simple JWT -
for ~~global~~ globaly-

Ex- · settings.py -

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    )
}
```

· urls.py -

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [
    path('gettoken/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('refreshtoken/', TokenRefreshView.as_view(), name='token_refresh'),
]
```



- You can also include a route for Simple JWT's TokenVerifyView if you wish to allow API users to verify HMAC-signed tokens without having access to your signing key.

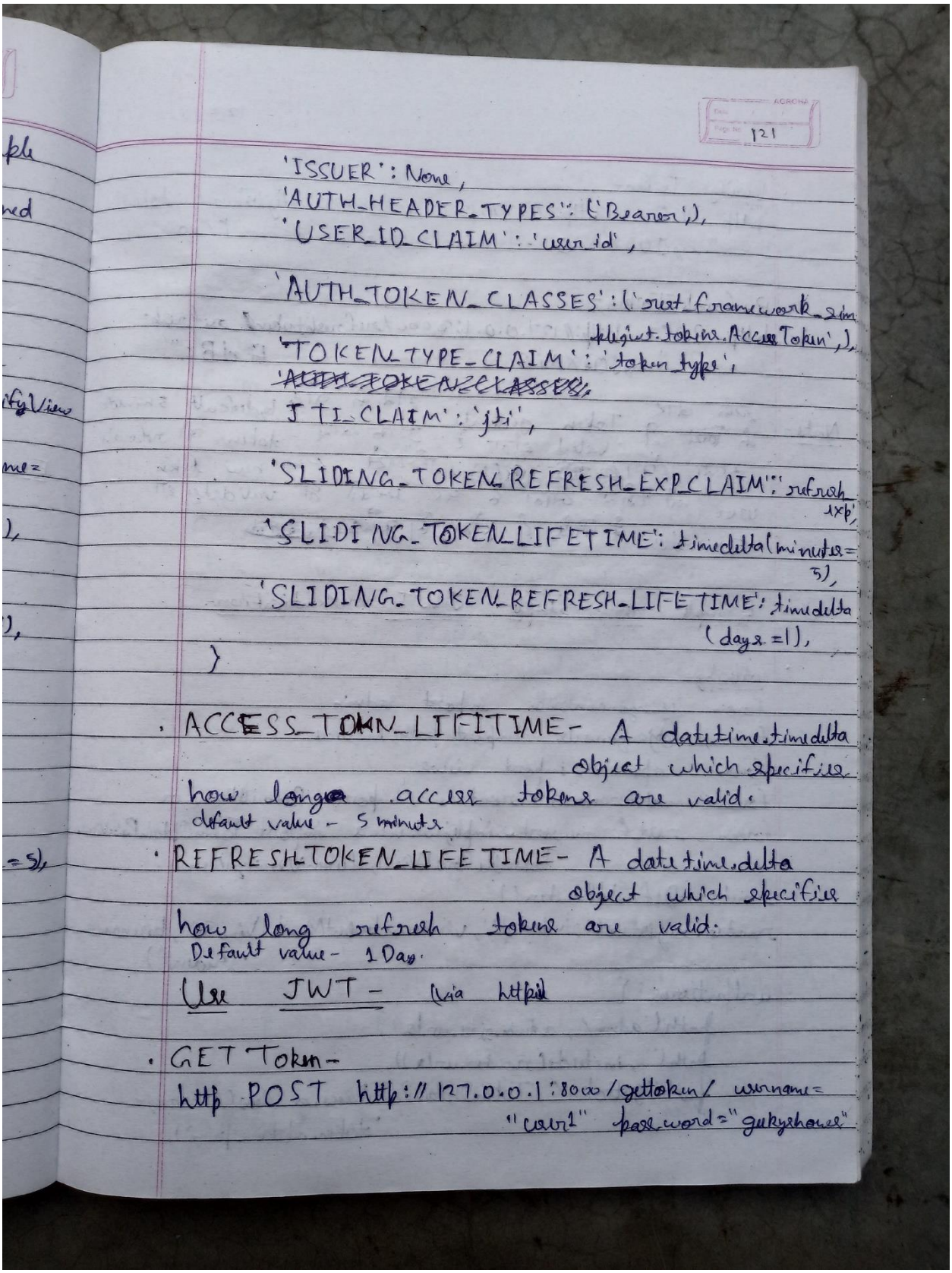
Ex

- urls.py -

```
from rest_framework_simplejwt.views import
        TokenObtainPairView, TokenRefreshView, TokenVerifyView
urlpatterns = [
    path('gettoken/', TokenObtainPairView.as_view(), name=
                                        'token_obtain_pair'),
    path('refreshtoken/', TokenRefreshView.as_view(),
                                    name='token_refresh'),
    path('verifytoken/', TokenVerifyView.as_view(),
                                    name='token_verify'),
]
```

JWT Default Settings -

```
from datetime import timedelta
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    'ROTATE_REFRESH_TOKENS': False,
    'BLACKLIST_AFTER_ROTATION': True,

    'ALGORITM': 'HS256',
    'SIGNINGKEY': settings.SECRET_KEY,
    'VERIFYING_KEY': None,
    'AUDIENCE': None,
```

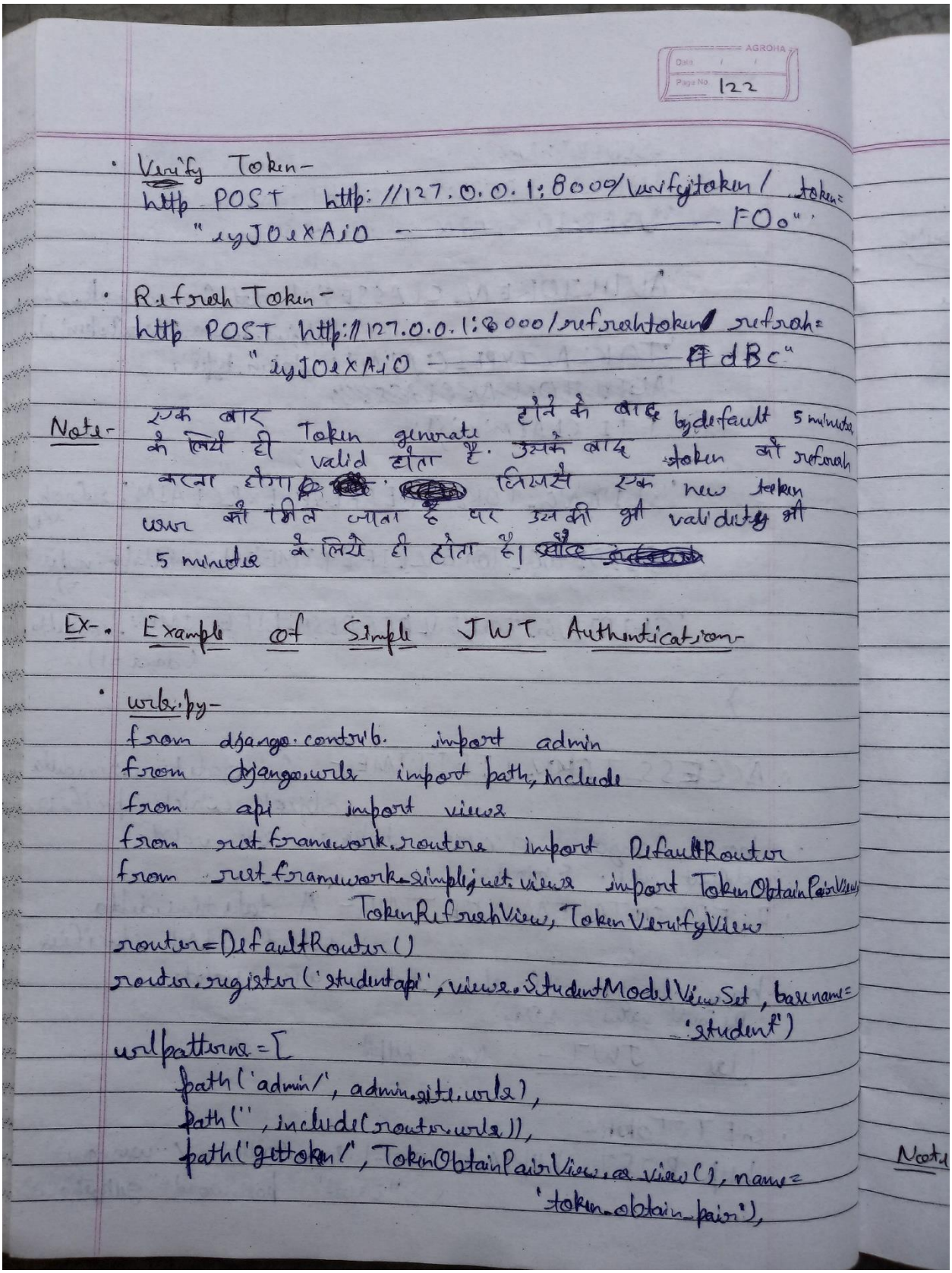


```
'ISSUER': None,
'AUTH_HEADER_TYPES': ('Bearer',),
'USER_ID_CLAIM': 'user_id',

'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
'TOKEN_TYPE_CLAIM': 'token_type',
'JTI_CLAIM': 'jti',

'SLIDING_TOKEN_REFRESH_EXP_CLAIM': 'refresh_exp',
'SLIDING_TOKEN_LIFETIME': timedelta(minutes=5),
'SLIDING_TOKEN_REFRESH_LIFETIME': timedelta(days=1),
}
```

- ACCESS_TOKEN_LIFITIME - A datetime.timedelta object which specifies how long access tokens are valid.
  default value - 5 minutes
- REFRESH_TOKEN_LIFETIME - A datetime.delta object which specifies how long refresh tokens are valid.
  Default value - 1 Day.

Use JWT - (via httpie)

- GET Token -
  http POST http://127.0.0.1:8000/gettoken/ username="user1" password="geekyshows"



- Verify Token-

http POST http://127.0.0.1:8000/verifytoken/ token="eyJ0eXAiO ---------- F0o"

- Refresh Token-

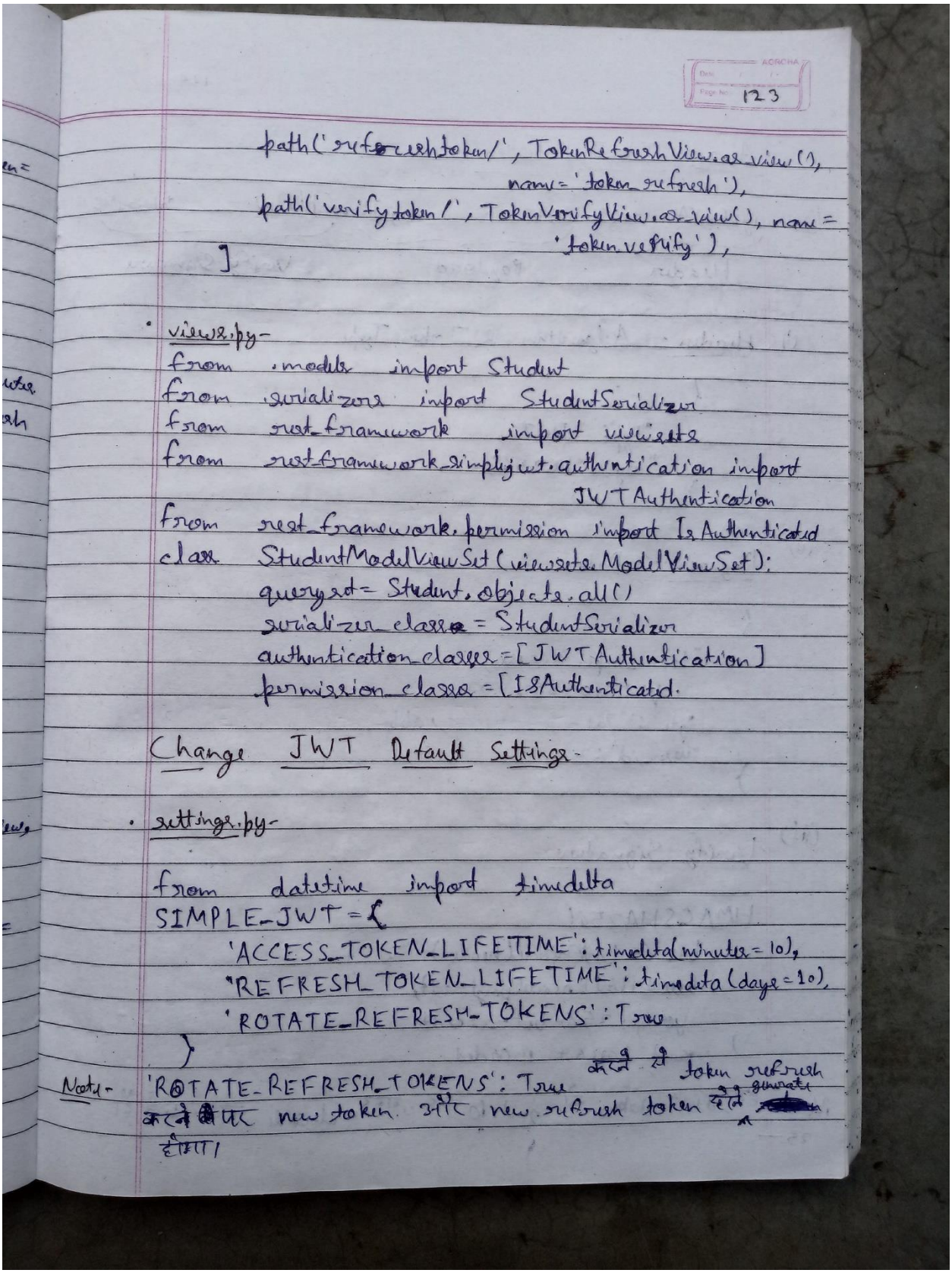
http POST http://127.0.0.1:8000/refreshtoken/ refresh="eyJ0eXAiO ---------- dBc"

Note- एक बार Token generate होने के बाद by default 5 minutes के लिये ही valid होता है. उसके बाद token को refresh करना होगा जिससे एक new token user को मिल जाता है पर उसकी भी validity भी 5 minutes के लिये ही होता है।

Ex- Example of Simple JWT Authentication-

- urls.py-

```
from django.contrib. import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
from rest_framework_simplejwt.views import TokenObtainPairView,
                    TokenRefreshView, TokenVerifyView
router=DefaultRouter()
router.register('studentapi', views.StudentModelViewSet, basename=
                                        'student')
urlpatterns=[
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
    path('gettoken/', TokenObtainPairView.as_view(), name=
                                        'token_obtain_pair'),
```



```
    path('refreshtoken/', TokenRefreshView.as_view(),
                                    name='token_refresh'),
    path('verifytoken/', TokenVerifyView.as_view(), name=
                                    'token_verify'),
]
```

- views.py-

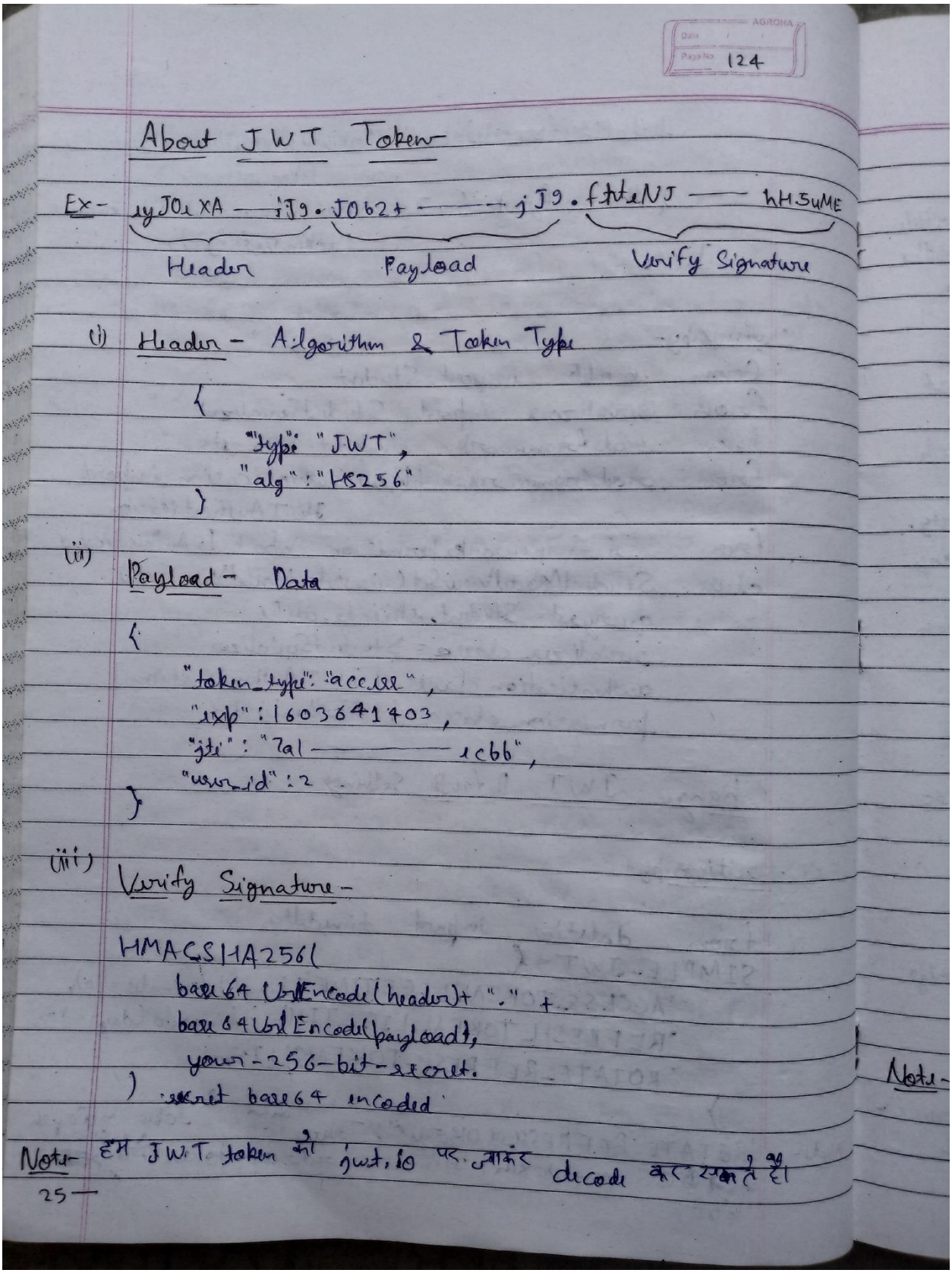
```
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
from rest_framework_simplejwt.authentication import
                                    JWTAuthentication
from rest_framework.permissions import IsAuthenticated
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [JWTAuthentication]
    permission_classes = [IsAuthenticated]
```

Change JWT Default Settings-

- settings.py-

```
from datetime import timedelta
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=10),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=10),
    'ROTATE_REFRESH_TOKENS': True
}
```

Note- 'ROTATE_REFRESH_TOKENS': True करने से token refresh करने पर new token और new refresh token देने लगता है।



AGROHA
Page No. 124

About JWT Token

Ex- eyJ0eXA ——— iJ9.JO62+ ——— jJ9.ftNeNJ ——— hH3uME

Header Payload Verify Signature

(i) Header - Algorithm & Token Type

```
{
    "typ": "JWT",
    "alg": "HS256"
}
```

(ii) Payload - Data

```
{
    "token_type": "access",
    "exp": 1603641403,
    "jti": "7a1 ——— 1c66",
    "user_id": 2
}
```

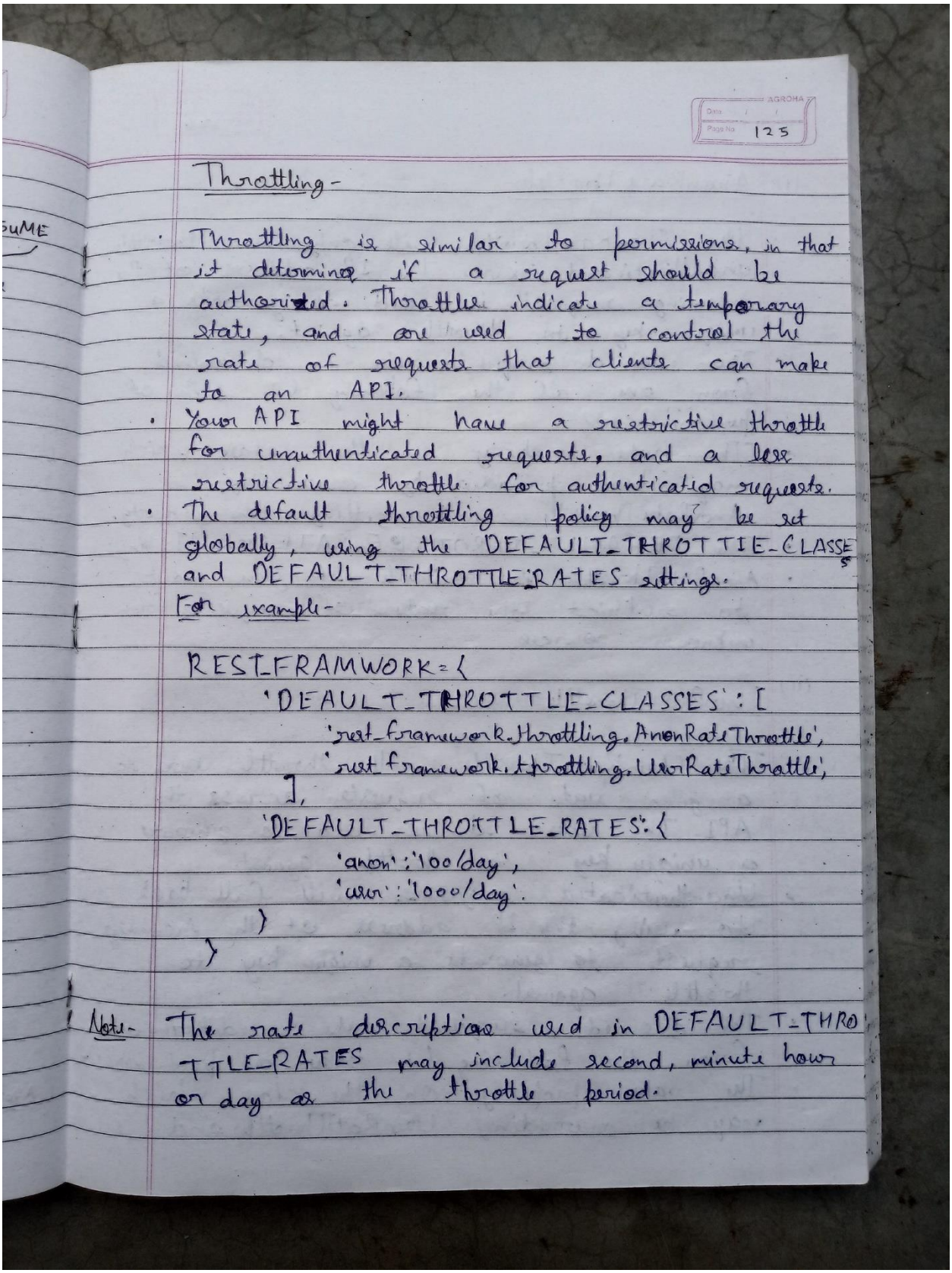
(iii) Verify Signature -

HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
your-256-bit-secret.
) secret base64 encoded

Note-

Note- इस JWT token को jwt.io पर जाकर decode कर सकते है।

25—



Throttling -

- Throttling is similar to permissions, in that it determines if a request should be authorized. Throttles indicate a temporary state, and are used to control the rate of requests that clients can make to an API.
- Your API might have a restrictive throttle for unauthenticated requests, and a less restrictive throttle for authenticated requests.
- The default throttling policy may be set globally, using the DEFAULT_THROTTLE_CLASSES and DEFAULT_THROTTLE_RATES settings.

For example -

```
REST_FRAMWORK = {
    'DEAULT_THROTTLE_CLASSES' : [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon':'100/day',
        'user':'1000/day'
    }
}
```

Note - The rate descriptions used in DEFAULT_THROTTLE_RATES may include second, minute hour or day as the throttle period.

(i) AnonRateThrottle -

- The AnonRateThrottle will only ever throttle unauthenticated users. The IP address of the incoming request is used to generate a unique key to throttle against.
- The allowed request rate is determined from one of the following (in order of preference).
- The rate property on the class, which may be provided by overriding AnonRateThrottle and setting the property.
- The DEFAULT_THROTTLE_RATES['anon'] setting.
- AnonRateThrottle is suitable if you want to restrict the rate of requests from unknown sources.

(ii) UserRateThrottle -

- The UserRateThrottle will throttle users to a given rate of requests across the API. The user id is used to generate a unique key to throttle against.
- Unauthenticated requests will fall back to using the IP address of the incoming request to generate a unique key to throttle against.
- The allowed request rate is determined from the following (in order of preference).
- The rate property on the class, which may be overriding UserRateThrottle and

(iii)

setting the property.
- The DEFAULT_THROTTLE_RATES['user'] setting.

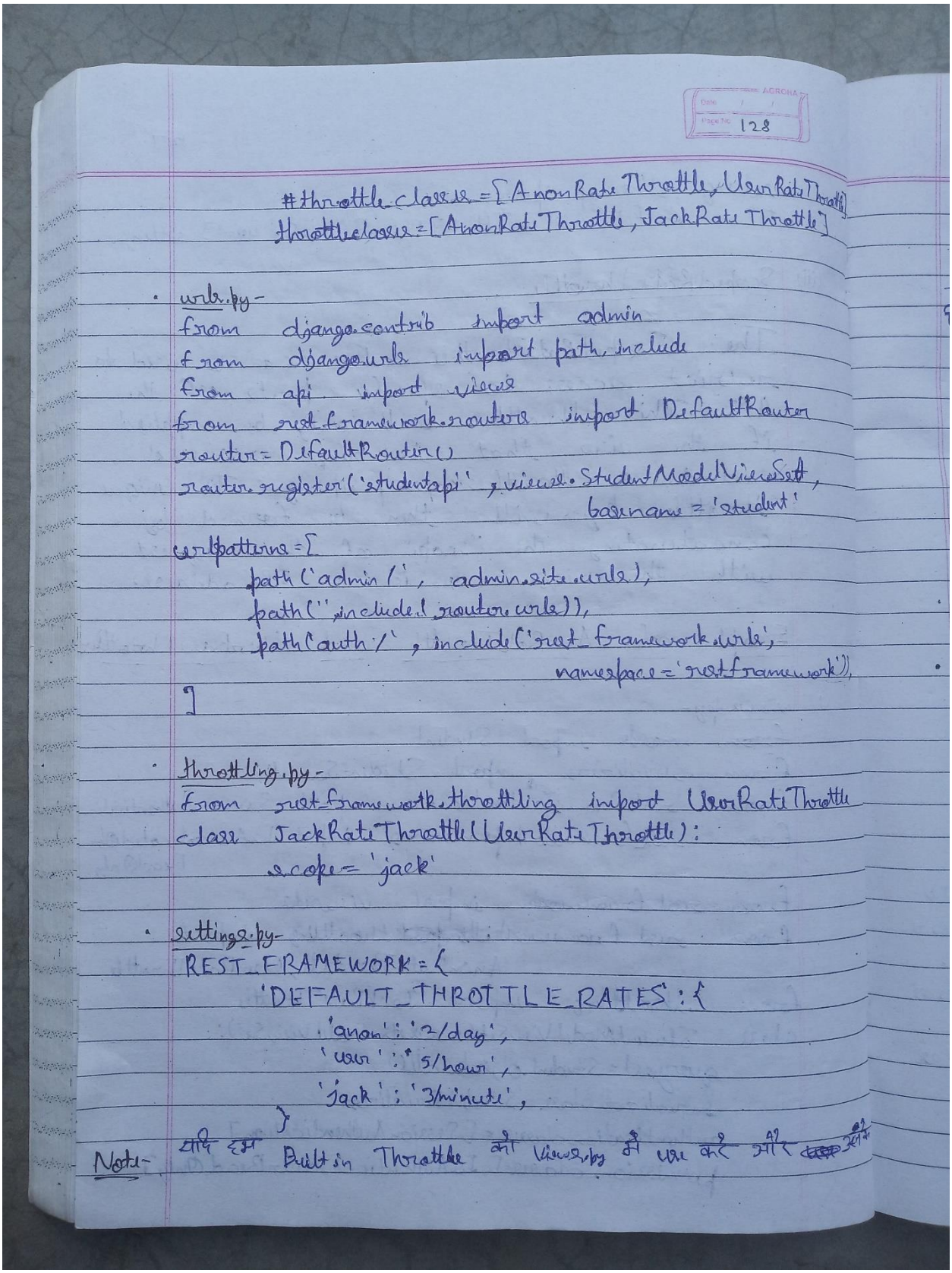
(iii) ScopedRateThrottle -

The ScopedRateThrottle class can be used to restrict access to specific parts of the API. This throttle will only be applied if the view that is being accessed includes a throttle_scope property. The unique throttle key will then be formed by concatenating the "scope" of the request with the unique user id or IP address.

Example of AnonRateThrottle and UserRateThrottle -

views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticatedOrReadOnly
from rest_framework import viewsets
from rest_framework.throttling import AnonRateThrottle, UserRateThrottle
from .throttling import JackRateThrottle
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticatedOrReadOnly]
```



#throttle_classes = [AnonRateThrottle, UserRateThrottle]
throttle_classes = [AnonRateThrottle, JackRateThrottle]

- urls.py -

```
from django.contrib import admin
from django.urls import path, include
from api import views
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register('studentapi', views.StudentModelViewSet,
                basename = 'student')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
    path('auth/', include('rest_framework.urls',
                          namespace='restframework')),
]
```

- throttling.py -

```
from rest_framework.throttling import UserRateThrottle
class JackRateThrottle(UserRateThrottle):
    scope = 'jack'
```

- settings.py -

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_RATES': {
        'anon': '2/day',
        'user': '5/hour',
        'jack': '3/minute',
    }
```

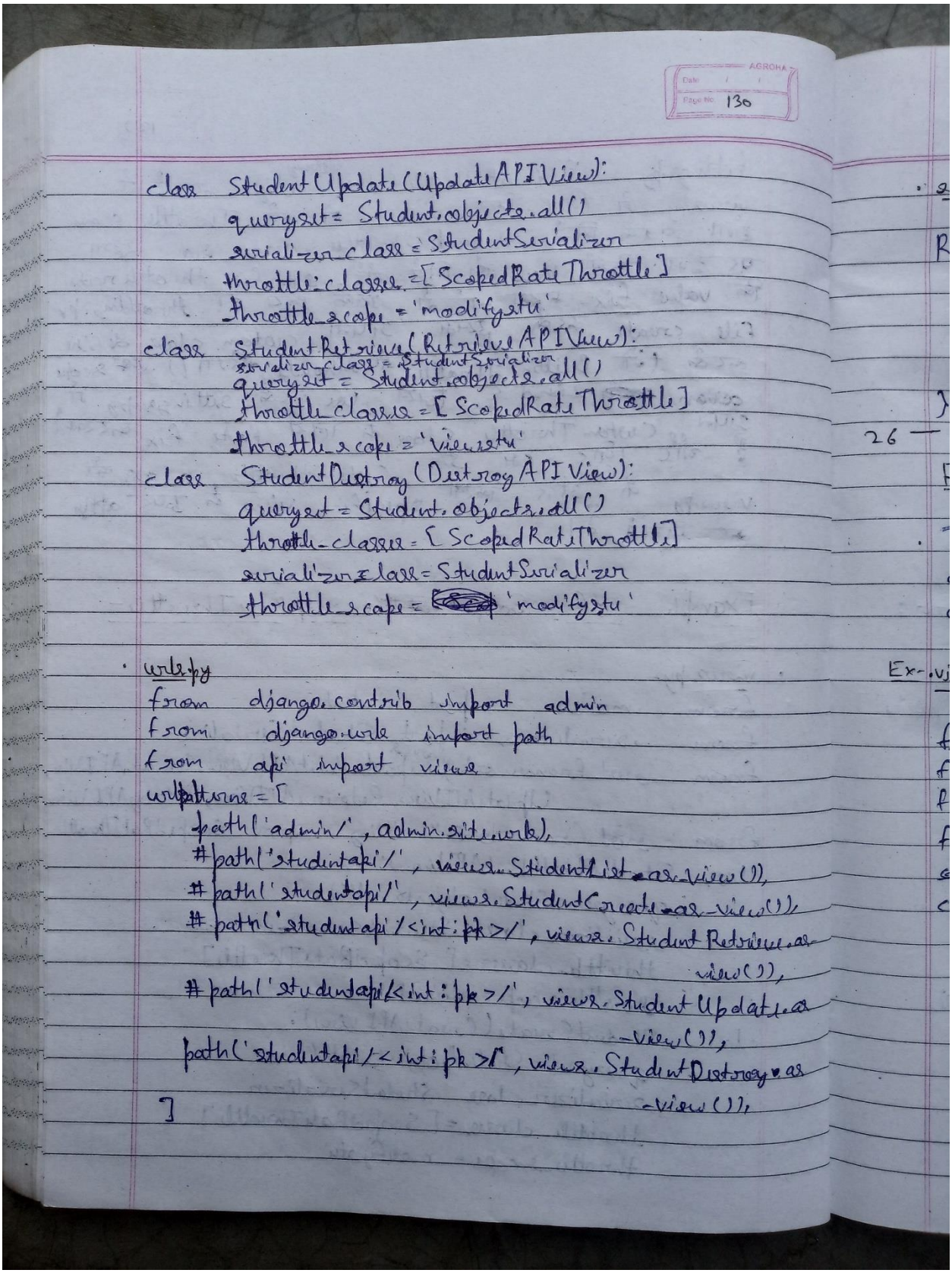
Note- Built in Throttle को views.py में use करें और

settings.py Default_Throttle_Rate globally assign कर दें तो views.py में जिस-जिस view में वो Throttle apply होगा सबके लिये Default throttle rate same रहेगा पर हम किसी view के लिये कोई specific throttle rate कि value fix करना चाहें तो उसके लिये हम throttling.py file create करके उसमें अपना custom class define करके (जो Builtin Throttle को inherit करेगा।) एक scope assign कर सकते हैं लिये और करके settings.py में अपने Custom Throttle class के लिये Rate fix कर सकते हैं और फिर हम उस Customthrottle class को views.py के अंदर किसी specific view के ऊपर apply कर सकते हैं।

- Example of ~~ScopedThrottle~~ ScopeRateThrottle -

- views.py -

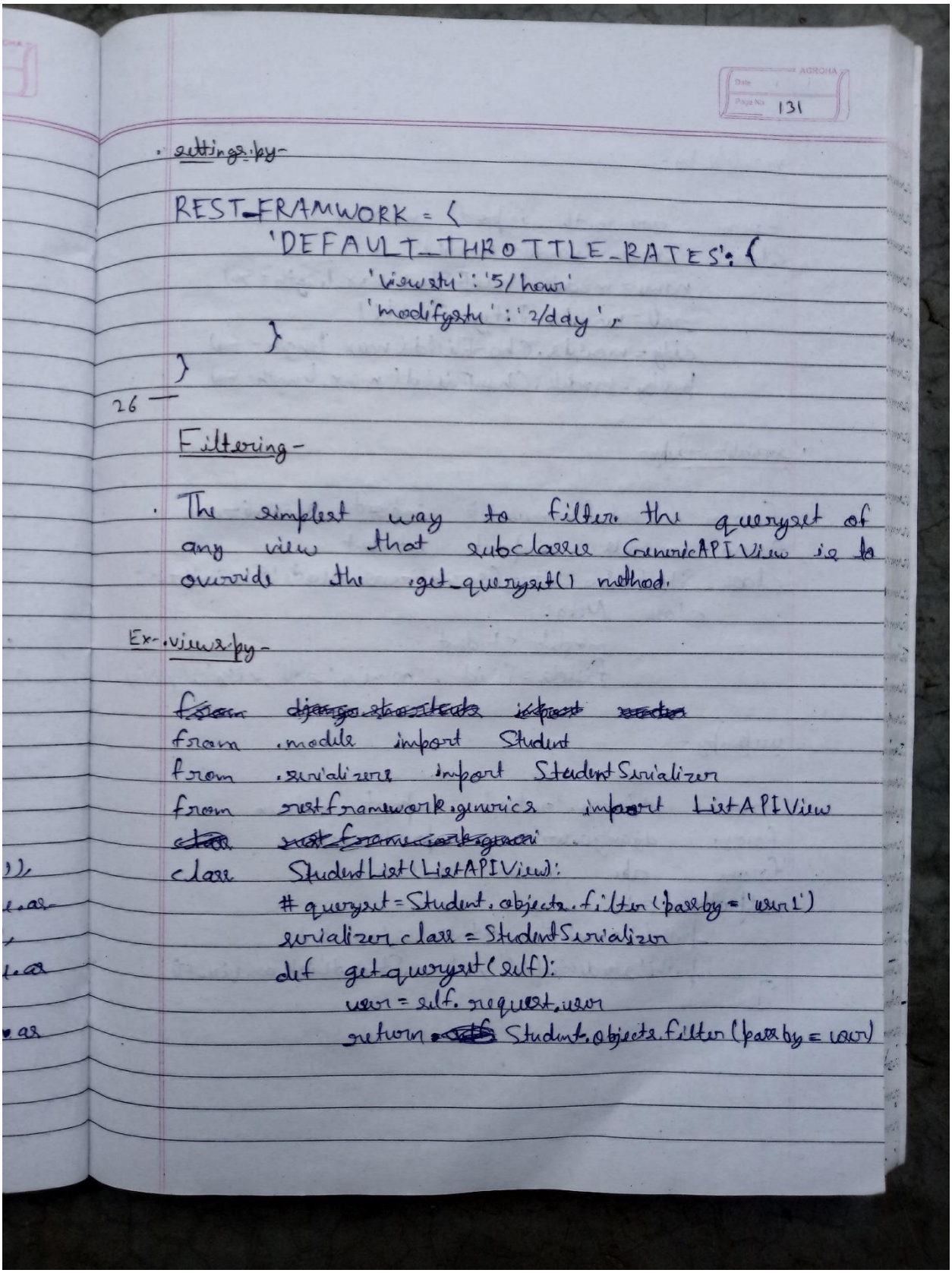
```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView, CreateAPIView,
                    UpdateAPIView, RetrieveAPIView, DestroyAPIView
from rest_framework.throttling import ScopedRateThrottle
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    throttle_classes = [ScopedRateThrottle]
    throttle_scope = 'viewstu'
class StudentCreate(CreateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    throttle_classes = [ScopedRateThrottle]
    throttle_scope = 'modifystu'
```



```
class StudentUpdate(UpdateAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    throttle_classes = [ScopedRateThrottle]
    throttle_scope = 'modifystu'
class StudentRetrieve(RetrieveAPIView):
    serializer_class = StudentSerializer
    queryset = Student.objects.all()
    throttle_classes = [ScopedRateThrottle]
    throttle_scope = 'viewstu'
class StudentDestroy(DestroyAPIView):
    queryset = Student.objects.all()
    throttle_classes = [ScopedRateThrottle]
    serializer_class = StudentSerializer
    throttle_scope = 'modifystu'
```

urls.py

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    #path('studentapi/', views.StudentList.as_view()),
    #path('studentapi', views.StudentCreate.as_view()),
    #path('studentapi/<int:pk>/', views.StudentRetrieve.as_view()),
    #path('studentapi/<int:pk>/', views.StudentUpdate.as_view()),
    path('studentapi/<int:pk>', views.StudentDestroy.as_view()),
]
```



Page No 131

· settings.py -

```
REST_FRAMWORK = {
    'DEFAULT_THROTTLE_RATES': {
        'viewstu': '5/hour'
        'modifystu': '2/day',
    }
}
```

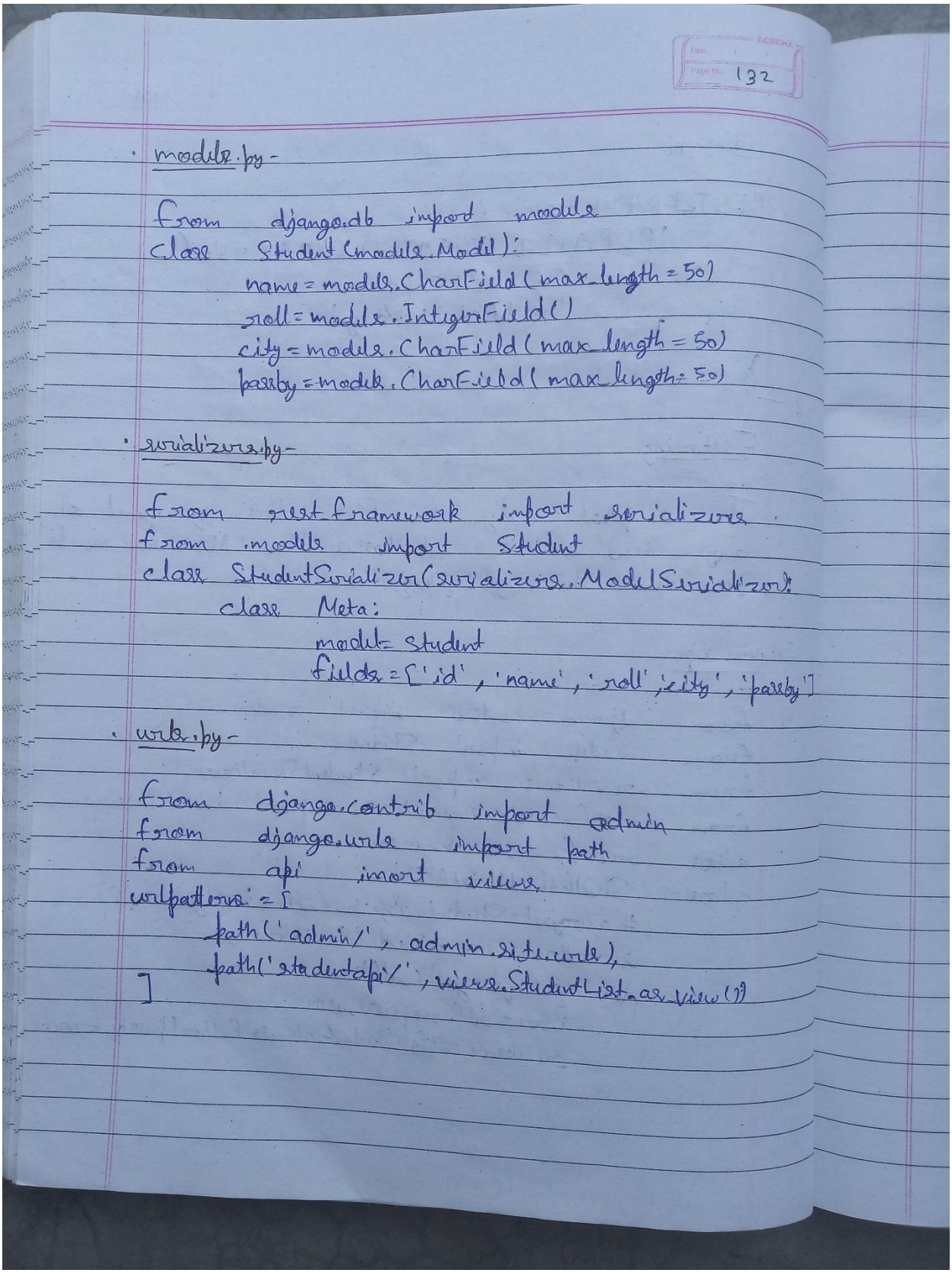
26 —

Filtering -

· The simplest way to filter the queryset of any view that subclasses GenericAPIView is to override the .get_queryset() method.

Ex- views.py -

```
from .module import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
class StudentList(ListAPIView):
    # queryset = Student.objects.filter(passby='user1')
    serializer_class = StudentSerializer
    def get_queryset(self):
        user = self.request.user
        return Student.objects.filter(passby=user)
```



Page No. 132

- models.py -

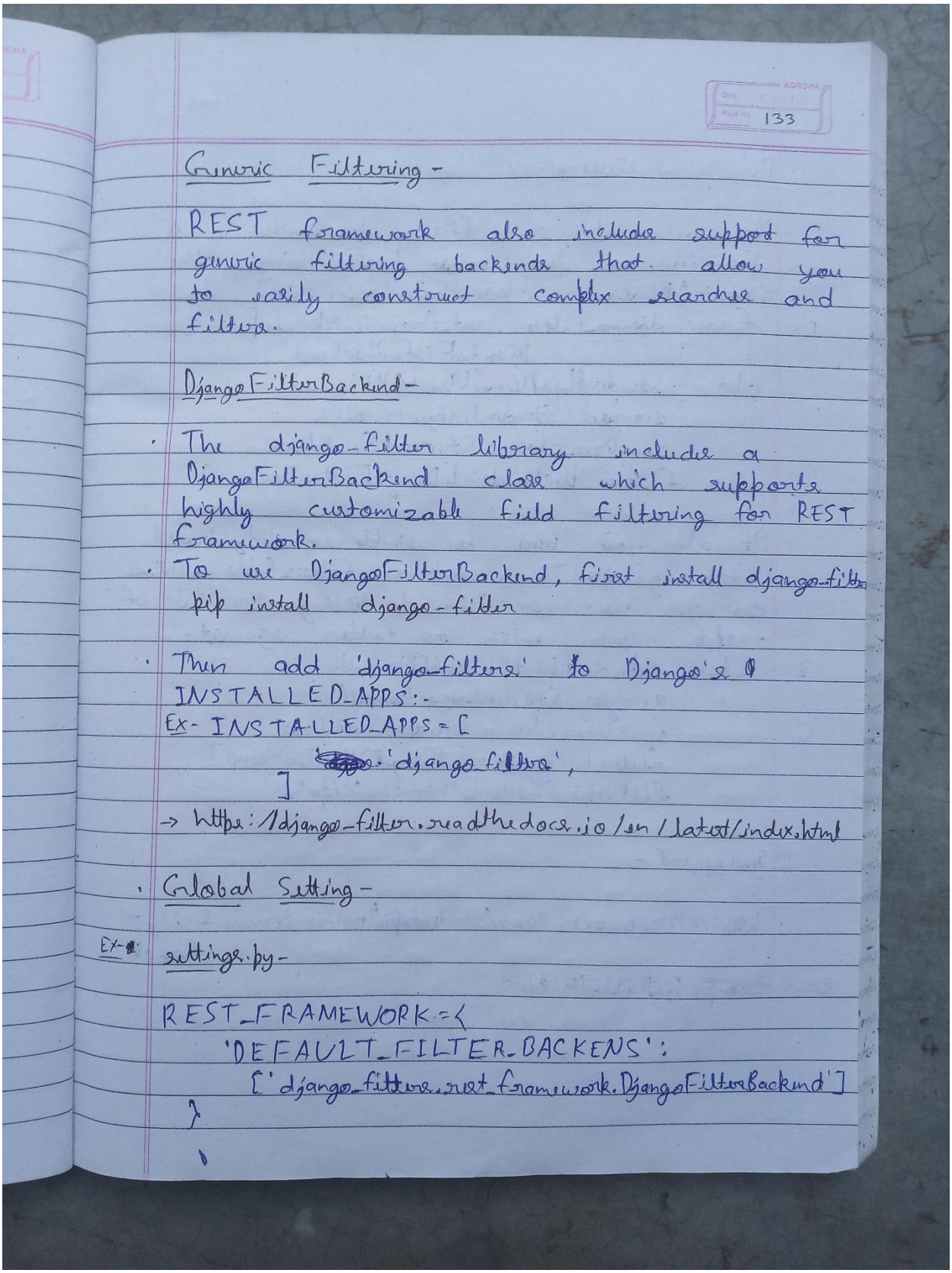
```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.IntegerField()
    city = models.CharField(max_length=50)
    passby = models.CharField(max_length=50)
```

- serializers.py -

```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll', 'city', 'passby']
```

- urls.py -

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/', views.StudentList.as_view())
]
```



Generic Filtering -

REST framework also include support for generic filtering backends that allow you to easily construct complex searches and filters.

DjangoFilterBackend -

- The django-filter library includes a DjangoFilterBackend class which supports highly customizable field filtering for REST framework.
- To use DjangoFilterBackend, first install django-filter
  pip install django-filter
- Then add 'django_filters' to Django's INSTALLED_APPS:-

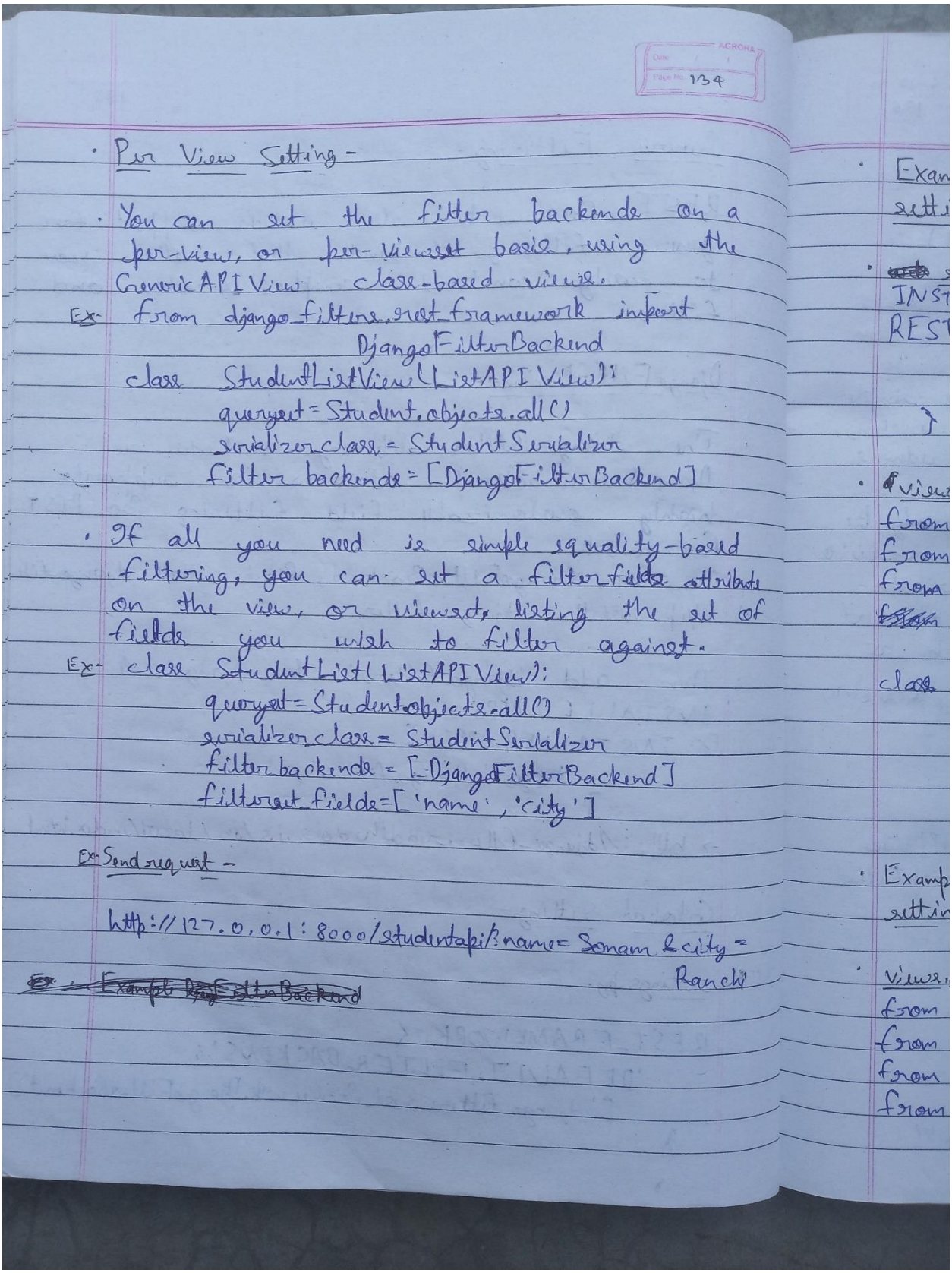
Ex- INSTALLED_APPS = [
    'django_filters',
]

-> https://django-filter.readthedocs.io/en/latest/index.html

- Global Setting -

Ex- settings.py -

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENS':
        ['django_filters.rest_framework.DjangoFilterBackend']
}
```



• Per View Setting -

• You can set the filter backends on a per-view, or per-viewset basis, using the GenericAPIView class-based views.

Ex- from django_filters.rest_framework import DjangoFilterBackend
class StudentListView(ListAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer
filter_backends = [DjangoFilterBackend]

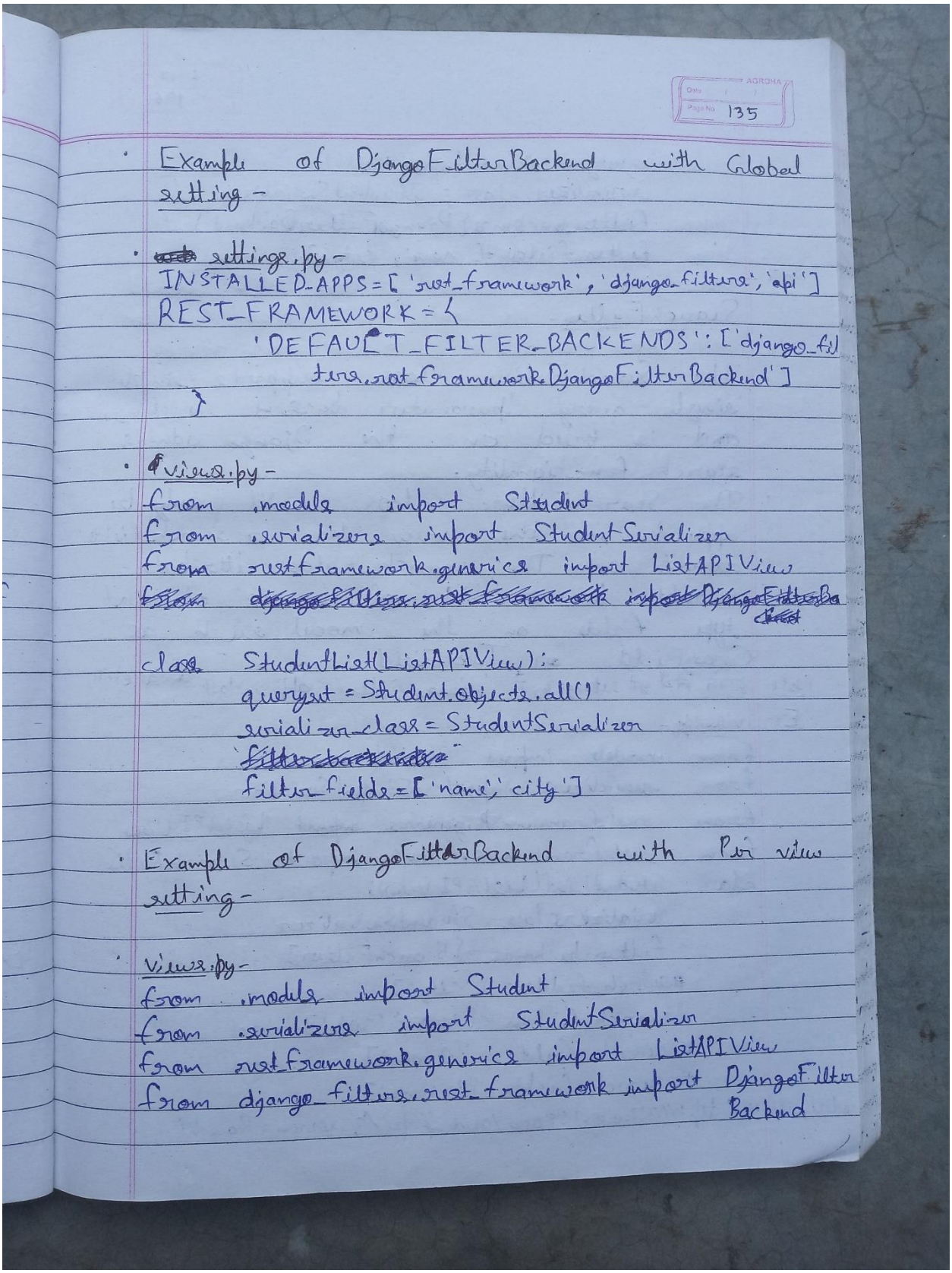
• If all you need is simple equality-based filtering, you can set a filterfields attribute on the view, or viewset, listing the set of fields you wish to filter against.

Ex- class StudentList(ListAPIView):
queryset = Student.objects.all()
serializer_class = StudentSerializer
filter_backends = [DjangoFilterBackend]
filterset_fields = ['name', 'city']

Ex- Send request -

http://127.0.0.1:8000/studentapi/?name=Sonam&city=Ranchi

~~Example DjangoFilterBackend~~



- Example of DjangoFilterBackend with Global setting -

- settings.py -

```
INSTALLED_APPS = ['rest_framework', 'django_filters', 'api']
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': ['django_filters.rest_framework.DjangoFilterBackend']
}
```

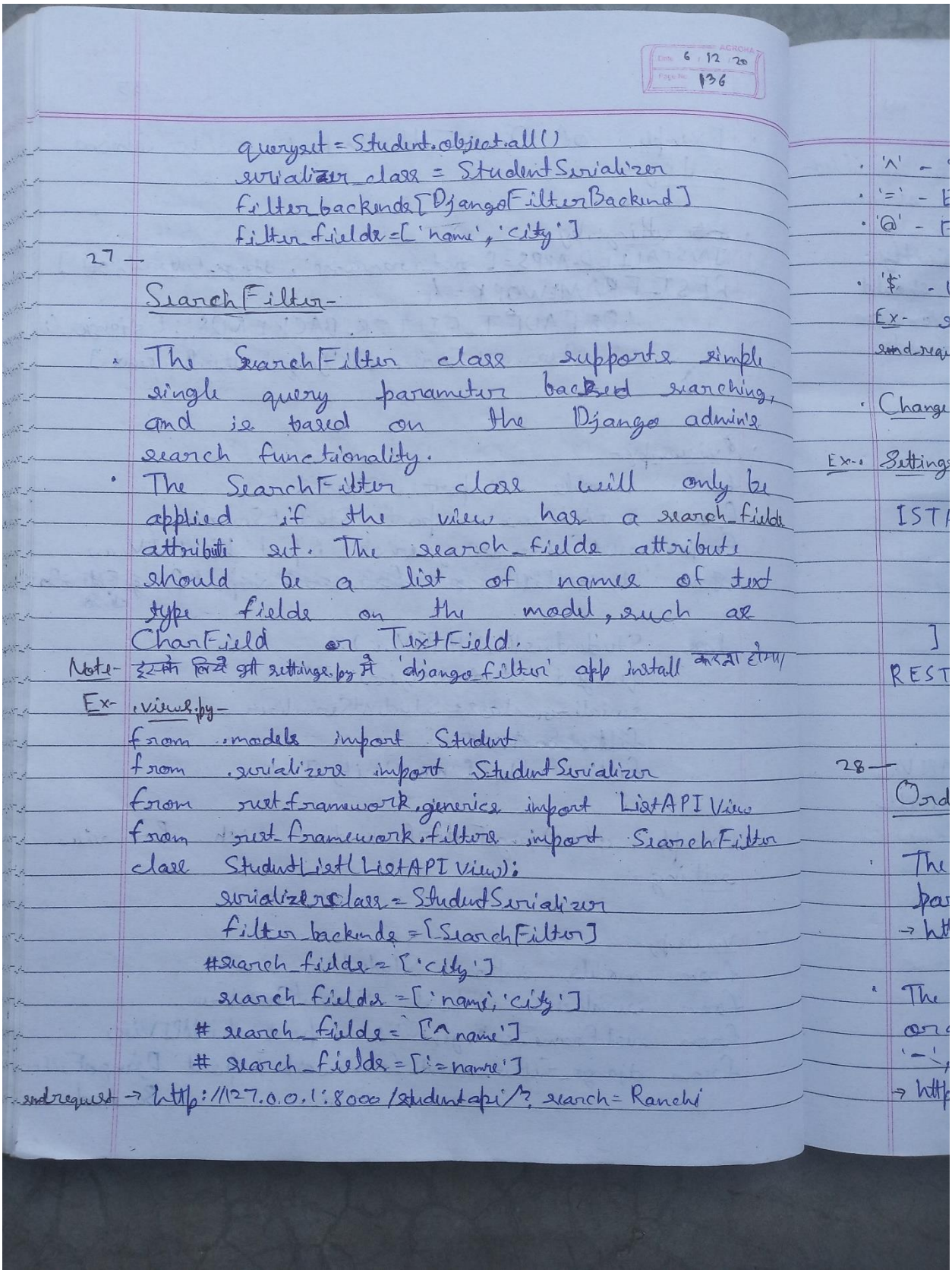
- views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    filter_fields = ['name', 'city']
```

- Example of DjangoFilterBackend with Per view setting -

- views.py -

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
from django_filters.rest_framework import DjangoFilterBackend
```



queryset = Student.objects.all()
serializer_class = StudentSerializer
filter_backends [DjangoFilterBackend]
filter_fields = ['name', 'city']

27 —

SearchFilter-

- The SearchFilter class supports simple single query parameter based searching, and is based on the Django admin's search functionality.
- The SearchFilter class will only be applied if the view has a search_fields attribute set. The search_fields attribute should be a list of names of text type fields on the model, such as CharField or TextField.

Note- इसके लिये get settings.py में 'django_filter' app install करना होगा।

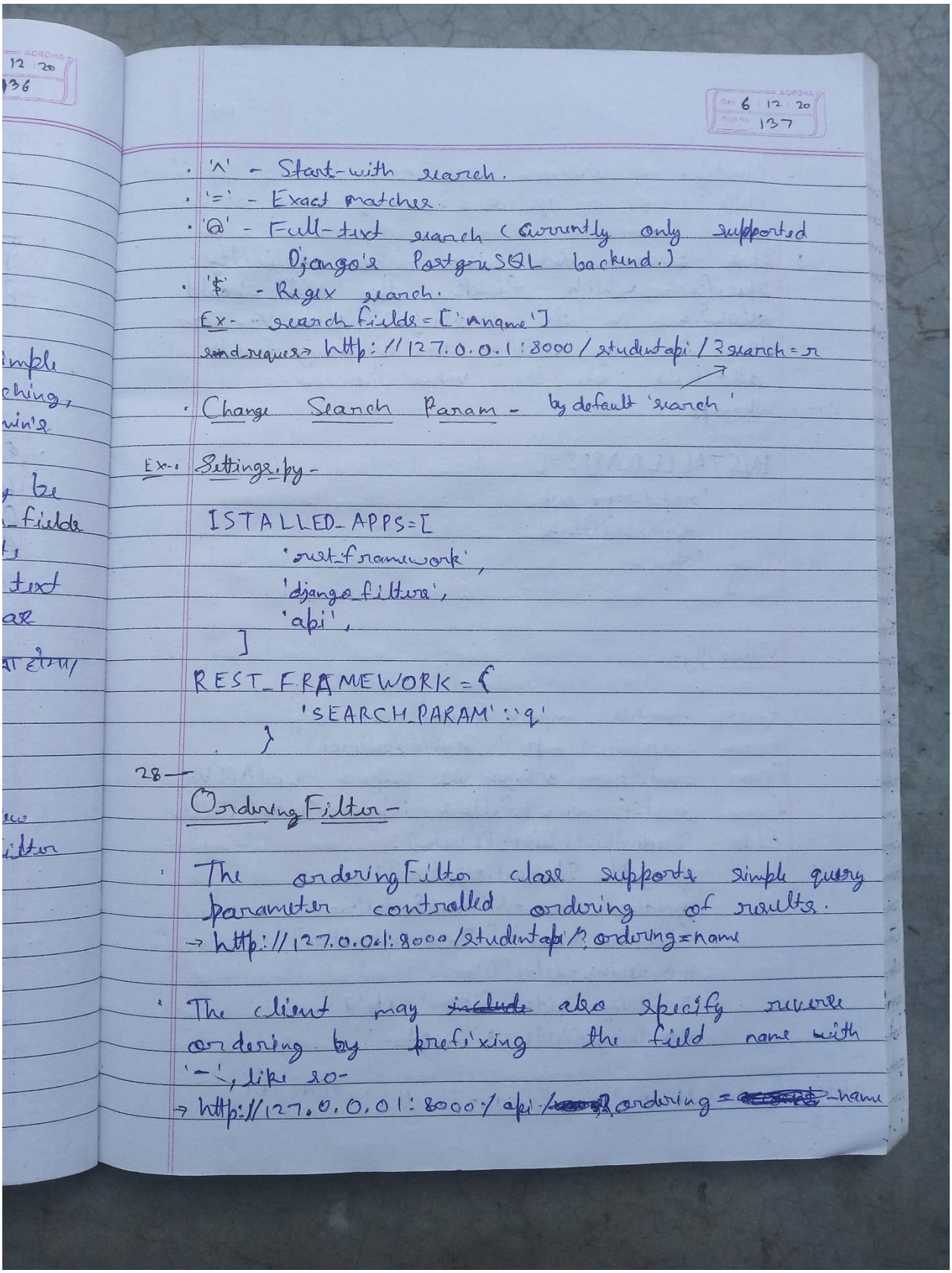
Ex- views.py-

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
from rest_framework.filters import SearchFilter
class StudentList(ListAPIView):
    serializer_class = StudentSerializer
    filter_backends = [SearchFilter]
    #search_fields = ['city']
    search_fields = ['name', 'city']
    # search_fields = ['^name']
    # search_fields = ['=name']
```

endrequest → http://127.0.0.1:8000/studentapi/?search=Ranchi

28 —

Ord



- '^' - Start-with search.
- '=' - Exact matches.
- '@' - Full-text search (Currently only supported Django's PostgreSQL backend.)
- '$' - Regex search.

Ex- search_fields = ['name']
send request → http://127.0.0.1:8000/studentapi/?search=r

- Change Search Param - by default 'search'

Ex:- Settings.py -

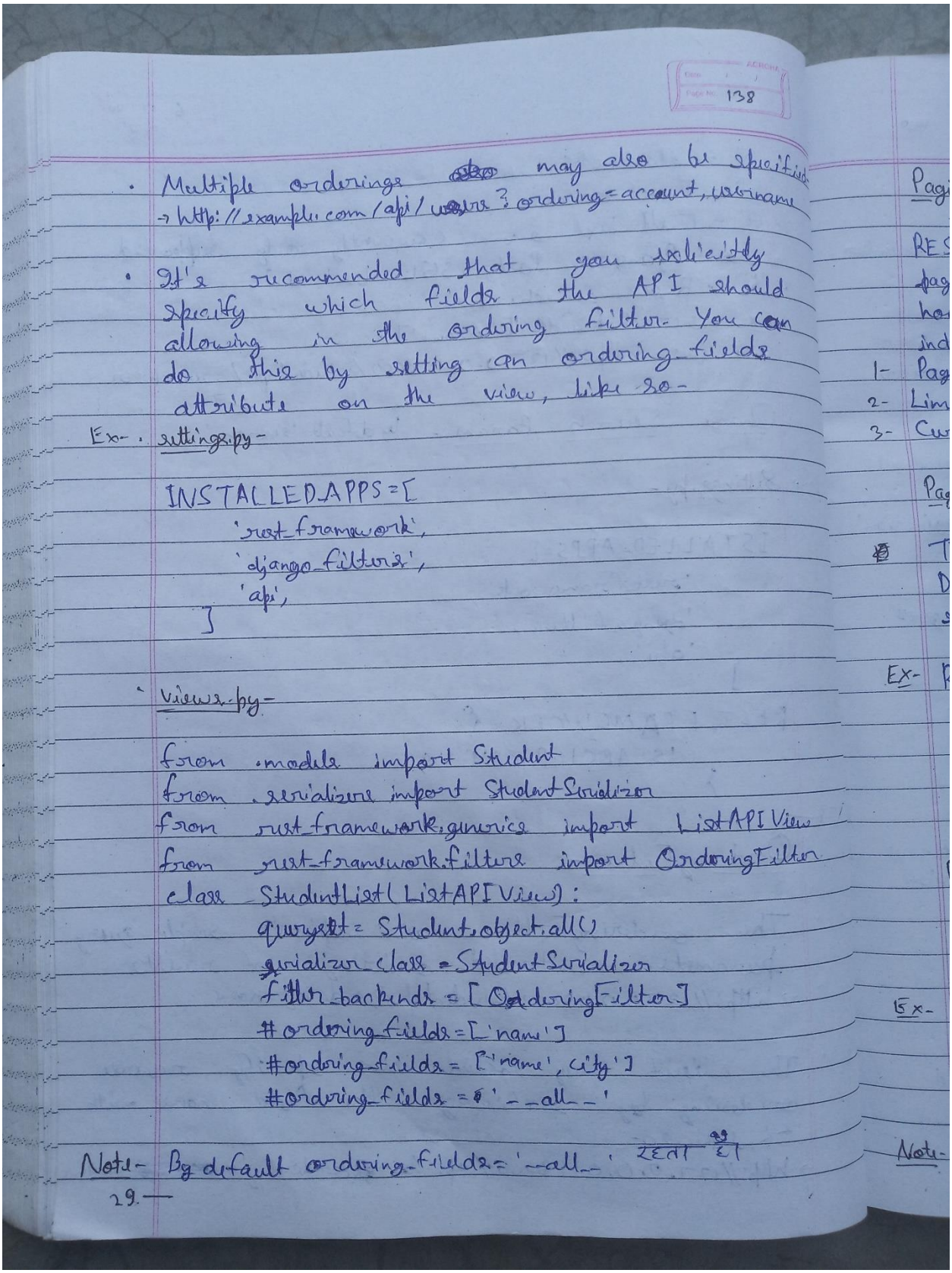
```
ISTALLED_APPS=[
        'rest_framework',
        'django_filters',
        'api',
    ]
REST_FRAMEWORK={
        'SEARCH_PARAM':'q'
    }
```

28—

Ordering Filter -

- The orderingFilter class supports simple query parameter controlled ordering of results.
→ http://127.0.0.1:8000/studentapi/?ordering=name

- The client may ~~include~~ also specify reverse ordering by prefixing the field name with '-', like so-
→ http://127.0.0.1:8000/api/?ordering=-name



- Multiple orderings may also be specified
→ http://example.com/api/users?ordering=account,username

- It's recommended that you explicitly specify which fields the API should allowing in the ordering filter. You can do this by setting an ordering_fields attribute on the view, like so-

Ex- . settings.py-

```
INSTALLED_APPS=[
    'rest_framework',
    'django_filters',
    'api',
]
```

- views.py-

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
from rest_framework.filters import OrderingFilter
class StudentList(ListAPIView):
    queryset = Student.object.all()
    serializer_class = StudentSerializer
    filter_backends = [OrderingFilter]
    #ordering_fields=['name']
    #ordering_fields = ['name', 'city']
    #ordering_fields = '__all__'
```

Note- By default ordering_fields='__all__'

29.—

Pagination-

REST framework includes support for customizable pagination styles. This allows you to modify how large result sets are split into individual pages of data.

1- PageNumberPagination
2- LimitOffsetPagination
3- CursorPagination

Pagination Global Setting-

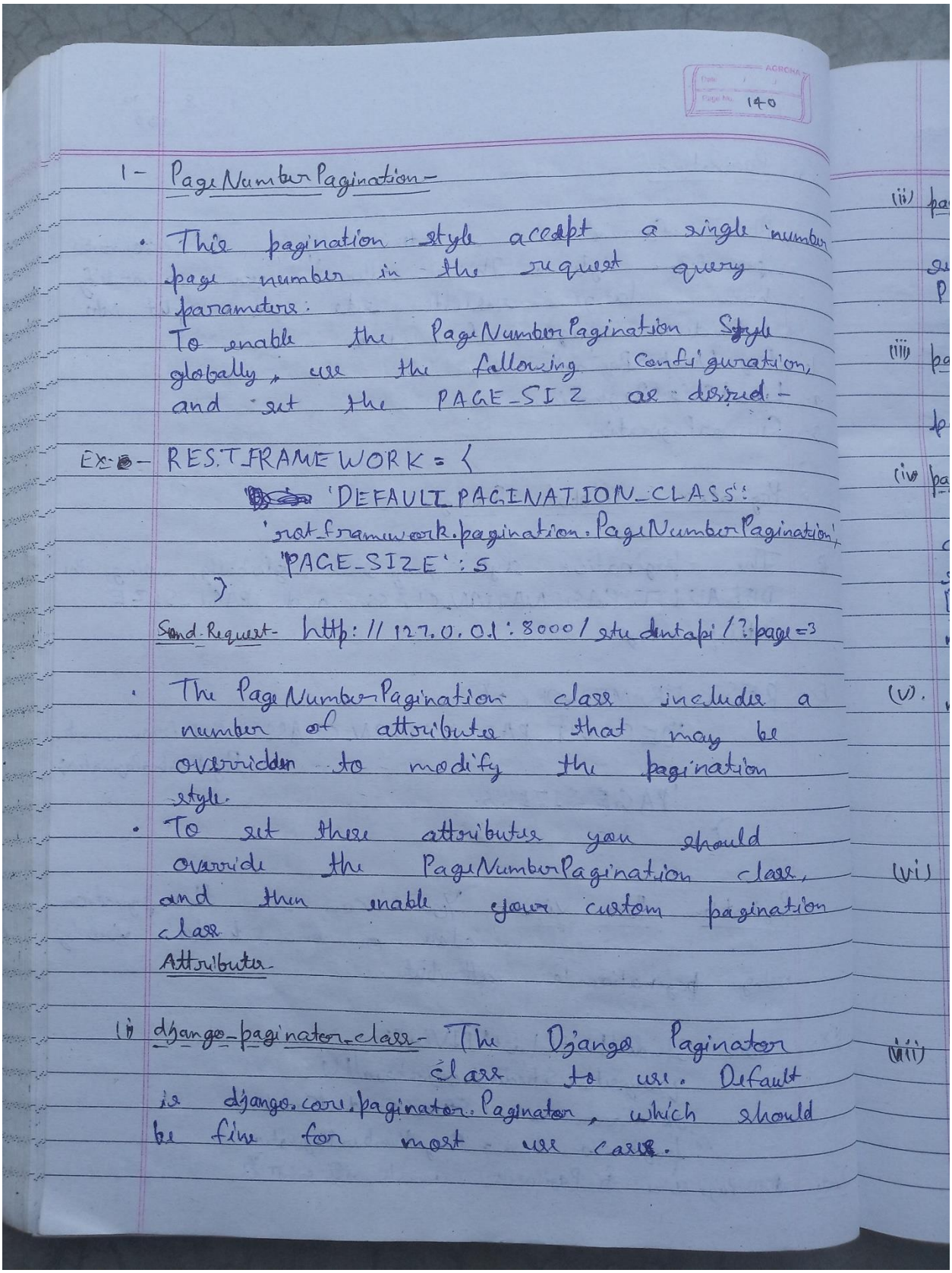
The pagination style may be set globally, using the DEFAULT_PAGINATION_CLASS and PAGE_SIZE setting keys.

Ex-
```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.
                                 PageNumberPagination',
    'PAGE_SIZE': 5
}
```

Pagination Per View - You can set the pagination class on an individual view by using pagination_class attribute.

Ex-
```
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    pagination_class = PageNumberPagination
```

Note- जब Page size न Pagination work नहीं करता है।



1- PageNumberPagination-

- This pagination style accept a single 'number' page number in the request query parameters.
To enable the PageNumberPagination style globally, use the following configuration, and set the PAGE_SIZ as desired:-

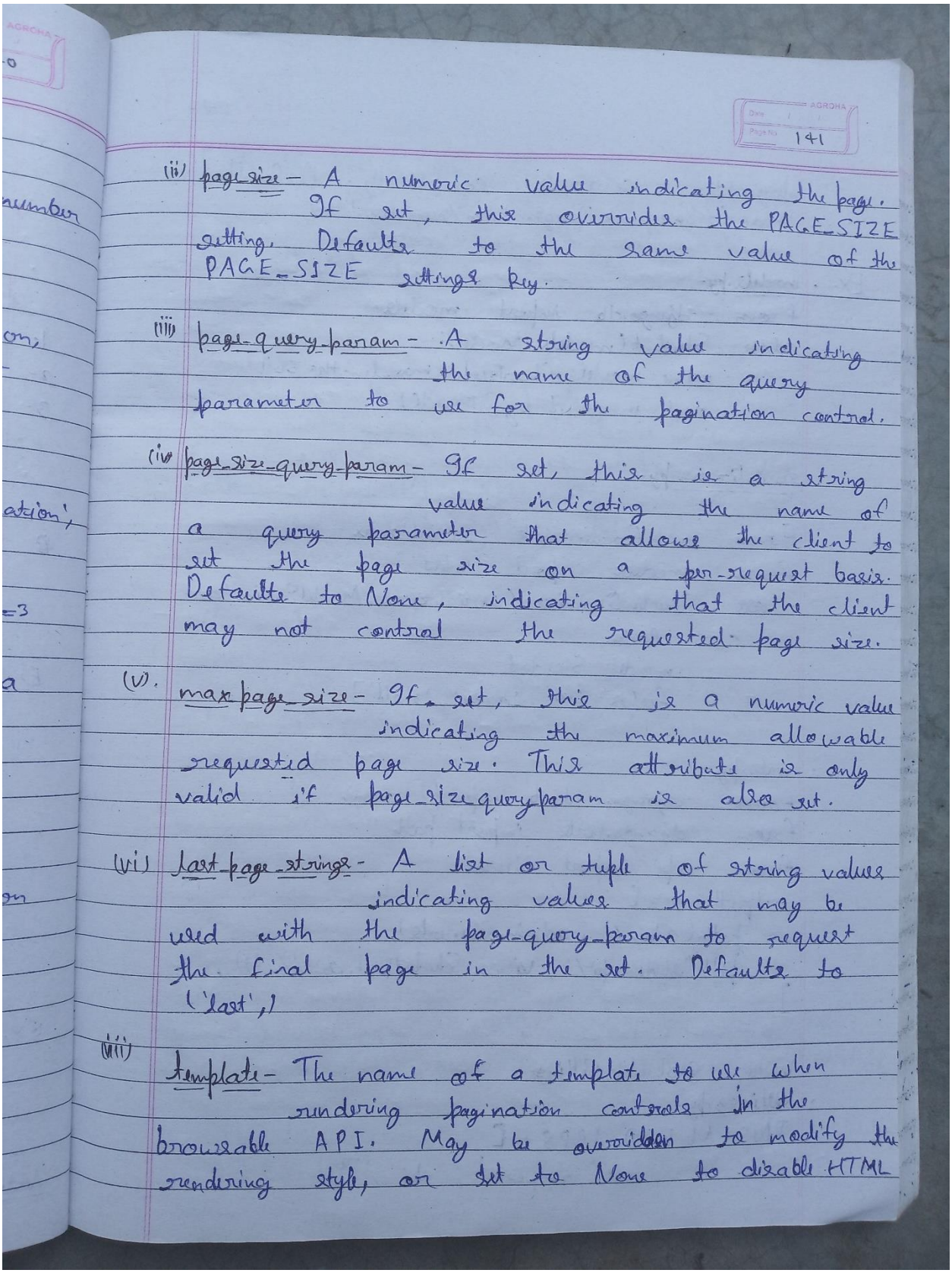
Ex- REST_FRAMEWORK = {
'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
'PAGE_SIZE': 5
}

Send Request- http://127.0.0.1:8000/studentapi/?page=3

- The PageNumberPagination class includes a number of attributes that may be overridden to modify the pagination style.
- To set these attributes you should override the PageNumberPagination class, and then enable your custom pagination class.

Attributes

(i) django_paginator_class- The Django Paginator class to use. Default is django.core.paginator.Paginator, which should be fine for most use cases.



(ii) page_size - A numeric value indicating the page. If set, this overrides the PAGE_SIZE setting. Defaults to the same value of the PAGE_SIZE settings key.

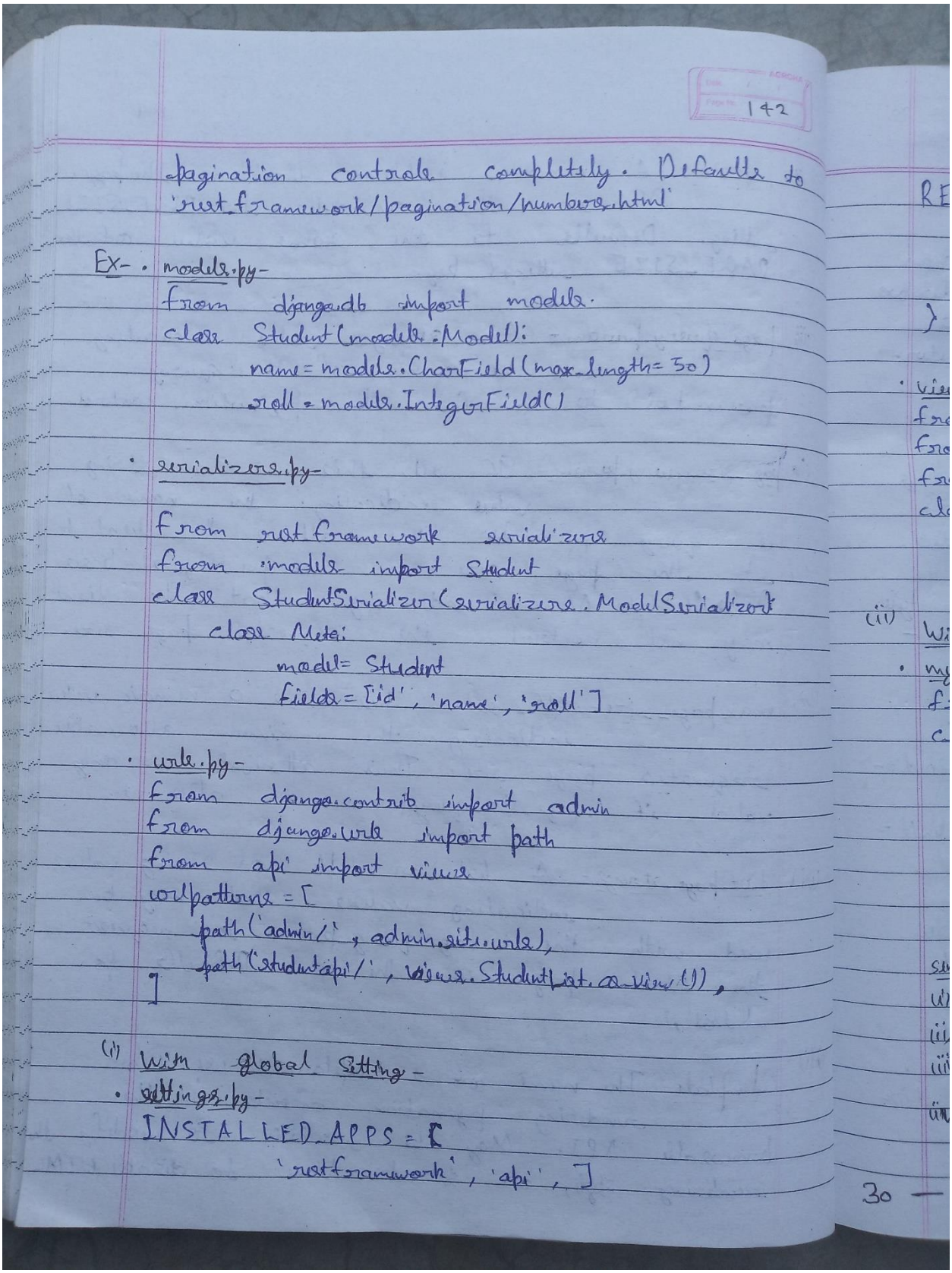
(iii) page_query_param - A string value indicating the name of the query parameter to use for the pagination control.

(iv) page_size_query_param - If set, this is a string value indicating the name of a query parameter that allows the client to set the page size on a per-request basis. Defaults to None, indicating that the client may not control the requested page size.

(v) max_page_size - If set, this is a numeric value indicating the maximum allowable requested page size. This attribute is only valid if page_size_query_param is also set.

(vi) last_page_strings - A list or tuple of string values indicating values that may be used with the page_query_param to request the final page in the set. Defaults to ('last',)

(vii) template - The name of a template to use when rendering pagination controls in the browsable API. May be overridden to modify the rendering style, or set to None to disable HTML



pagination controls completely. Defaults to 'rest_framework/pagination/numbers.html'

Ex- • models.py-

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.IntegerField()
```

• serializers.py-

```
from rest_framework serializers
from .models import Student
class StudentSerializer(serializers.ModelSerializer
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll']
```

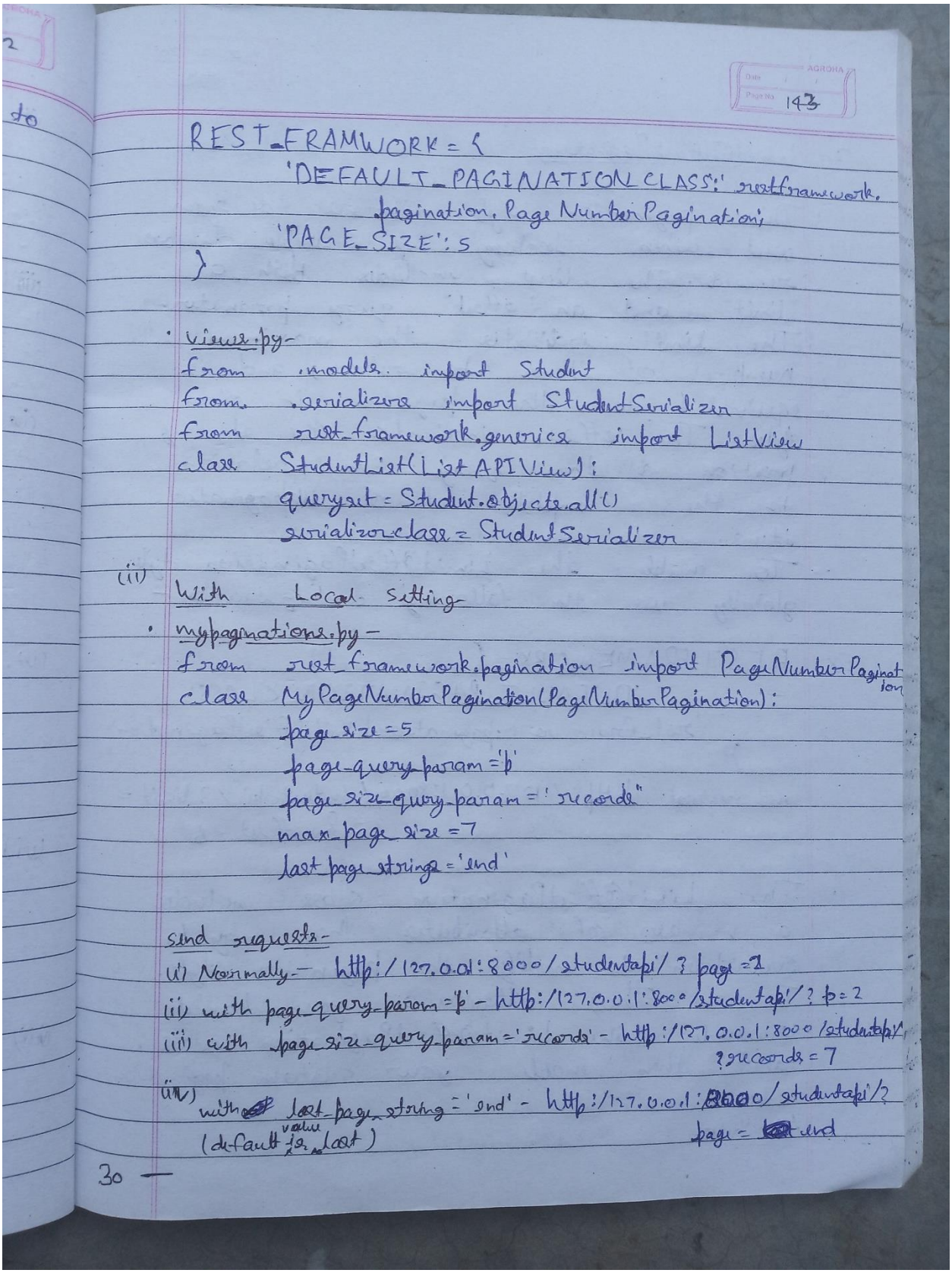
• urls.py-

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/', views.StudentList.as_view()),
]
```

(i) With global setting-

• settings.py-

```
INSTALLED_APPS = [
        'restframework', 'api', ]
```



```
REST_FRAMWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.
        pagination.PageNumberPagination',
    'PAGE_SIZE': 5
}
```

- views.py-

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListView
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

(ii) With Local setting-

- mypaginations.py -

```
from rest_framework.pagination import PageNumberPagination
class MyPageNumberPagination(PageNumberPagination):
    page_size = 5
    page_query_param = 'p'
    page_size_query_param = 'records'
    max_page_size = 7
    last_page_string = 'end'
```

Send requests-

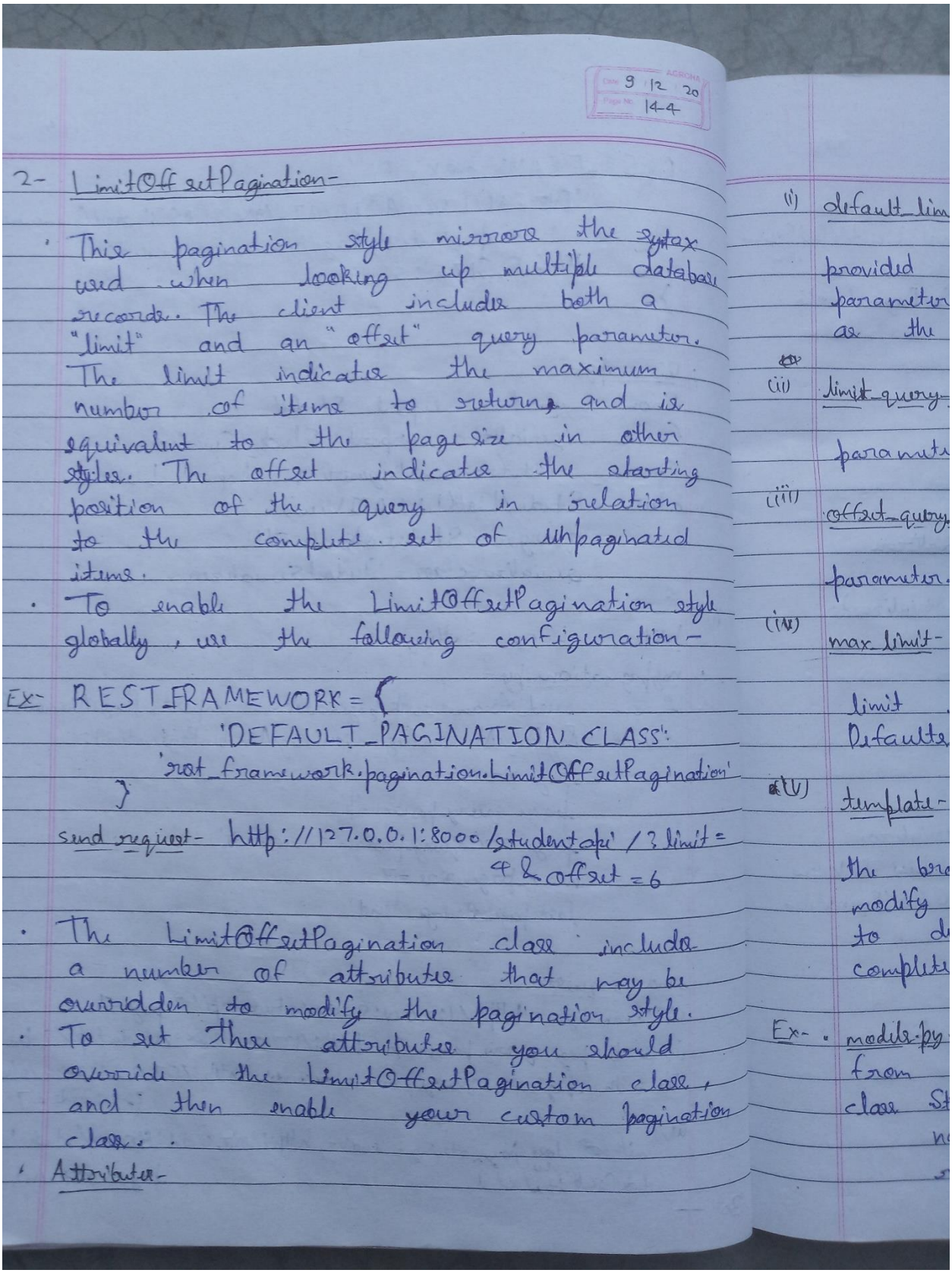
(i) Normally- http://127.0.0.1:8000/studentapi/?page=1

(ii) with page_query_param='p' - http://127.0.0.1:8000/studentapi/?p=2

(iii) with page_size_query_param='records' - http://127.0.0.1:8000/studentapi/?records=7

(iv) with last_page_string='end' - http://127.0.0.1:8000/studentapi/?page=end
(default value is last)

30



2- LimitOffsetPagination-

- This pagination style mirrors the syntax used when looking up multiple database records. The client includes both a "limit" and an "offset" query parameter. The limit indicates the maximum number of items to return, and is equivalent to the page size in other styles. The offset indicates the starting position of the query in relation to the complete set of unpaginated items.
- To enable the LimitOffsetPagination style globally, use the following configuration-

Ex- REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.LimitOffsetPagination'
}

send request- http://127.0.0.1:8000/studentapi/?limit=4&offset=6

- The LimitOffsetPagination class includes a number of attributes that may be overridden to modify the pagination style.
- To set these attributes you should override the LimitOffsetPagination class, and then enable your custom pagination class.
- Attributes-

(i) default_lim
provided
parameter
as the

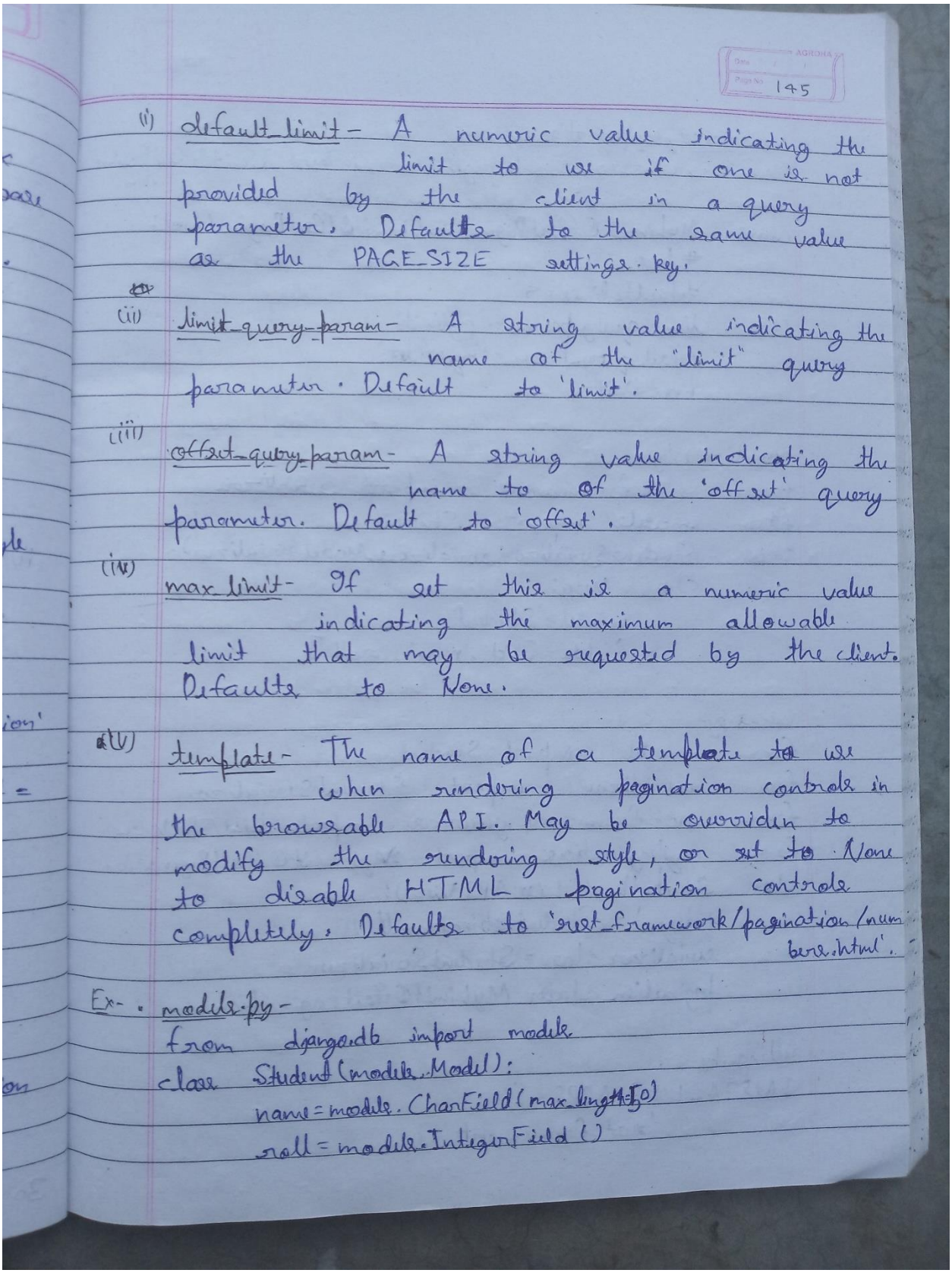
(ii) limit_query_
paramete

(iii) offset_query_
parameter.

(iv) max_limit-
limit
Defaults

(v) template-
the bro
modify
to d
complete

Ex- . module.py
from
class St
n



(i) default_limit - A numeric value indicating the limit to use if one is not provided by the client in a query parameter. Defaults to the same value as the PAGE_SIZE settings key.

(ii) limit_query_param - A string value indicating the name of the "limit" query parameter. Default to 'limit'.

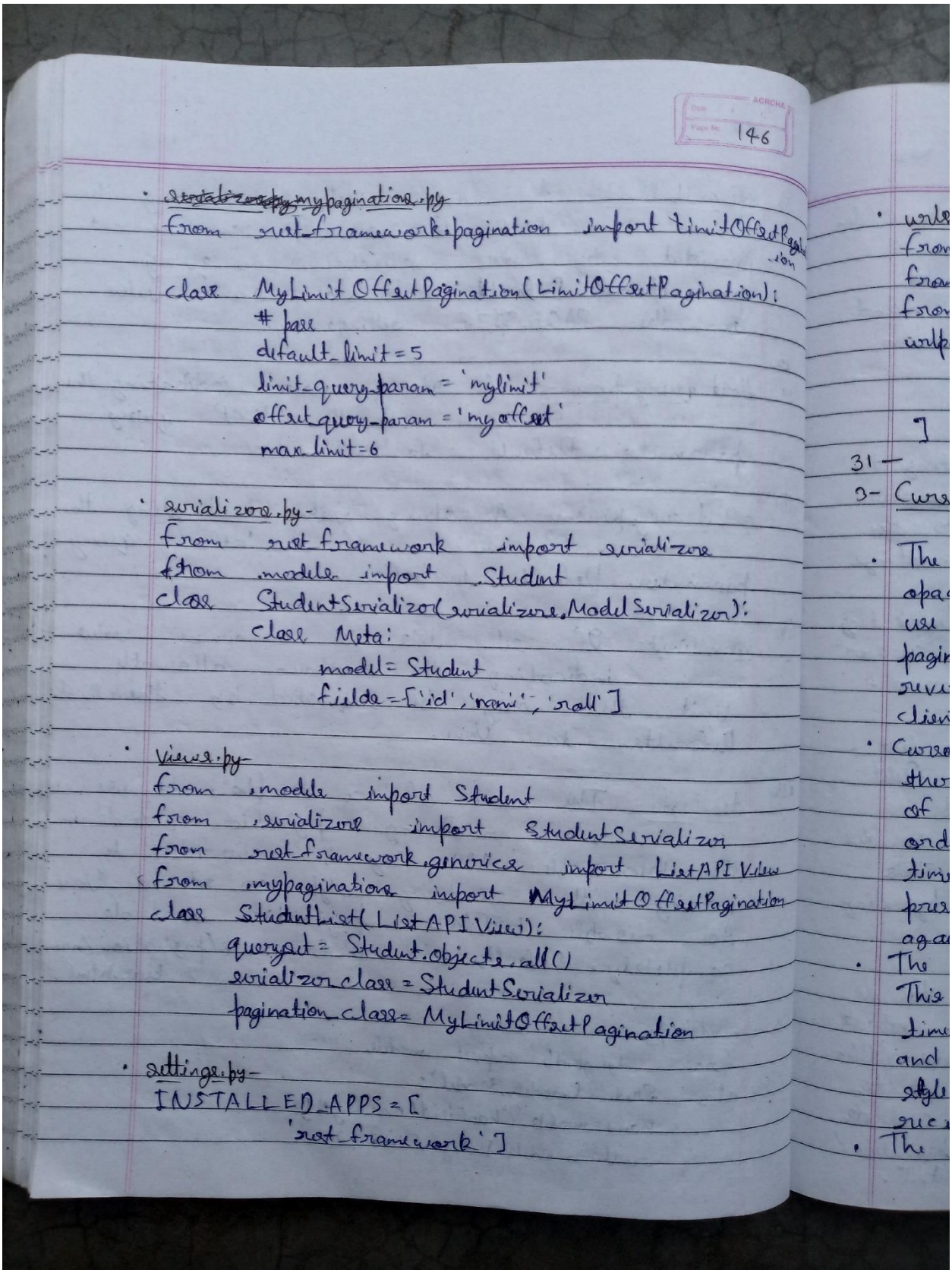
(iii) offset_query_param - A string value indicating the name to of the 'offset' query parameter. Default to 'offset'.

(iv) max_limit - If set this is a numeric value indicating the maximum allowable limit that may be requested by the client. Defaults to None.

(v) template - The name of a template to use when rendering pagination controls in the browsable API. May be overriden to modify the rendering style, or set to None to disable HTML pagination controls completely. Defaults to 'rest_framework/pagination/numbers.html'.

Ex. models.py -

```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.IntegerField()
```



- ~~serializers.py~~ mypagination.py

```
from rest_framework.pagination import LimitOffsetPagination

class MyLimitOffsetPagination(LimitOffsetPagination):
    # page
    default_limit = 5
    limit_query_param = 'mylimit'
    offset_query_param = 'myoffset'
    max_limit = 6
```

- serializers.py -

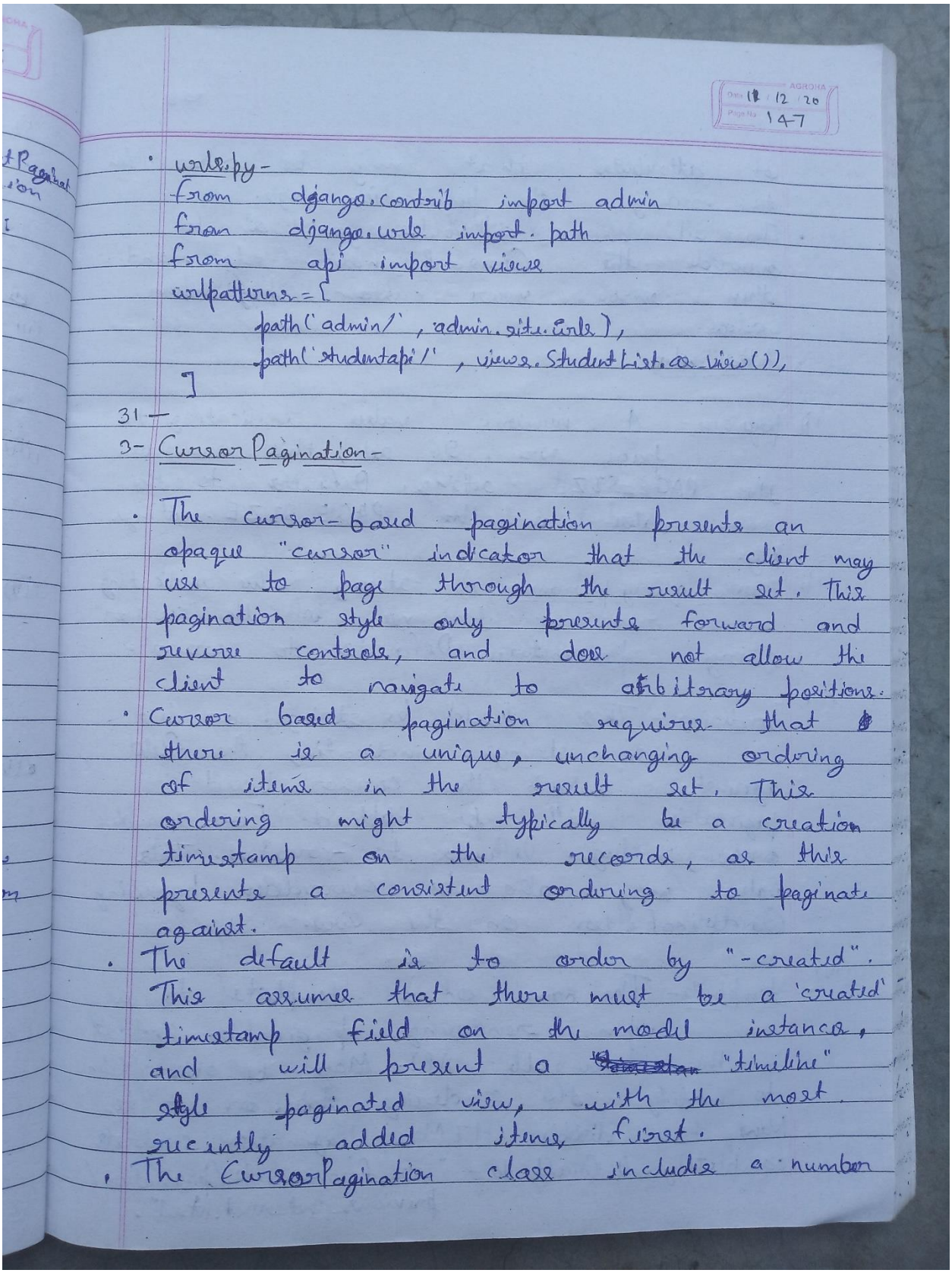
```
from rest_framework import serializers
from .models import Student
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll']
```

- views.py

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
from .mypagination import MyLimitOffsetPagination
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    pagination_class = MyLimitOffsetPagination
```

- settings.py -

```
INSTALLED_APPS = [
        'rest_framework']
```



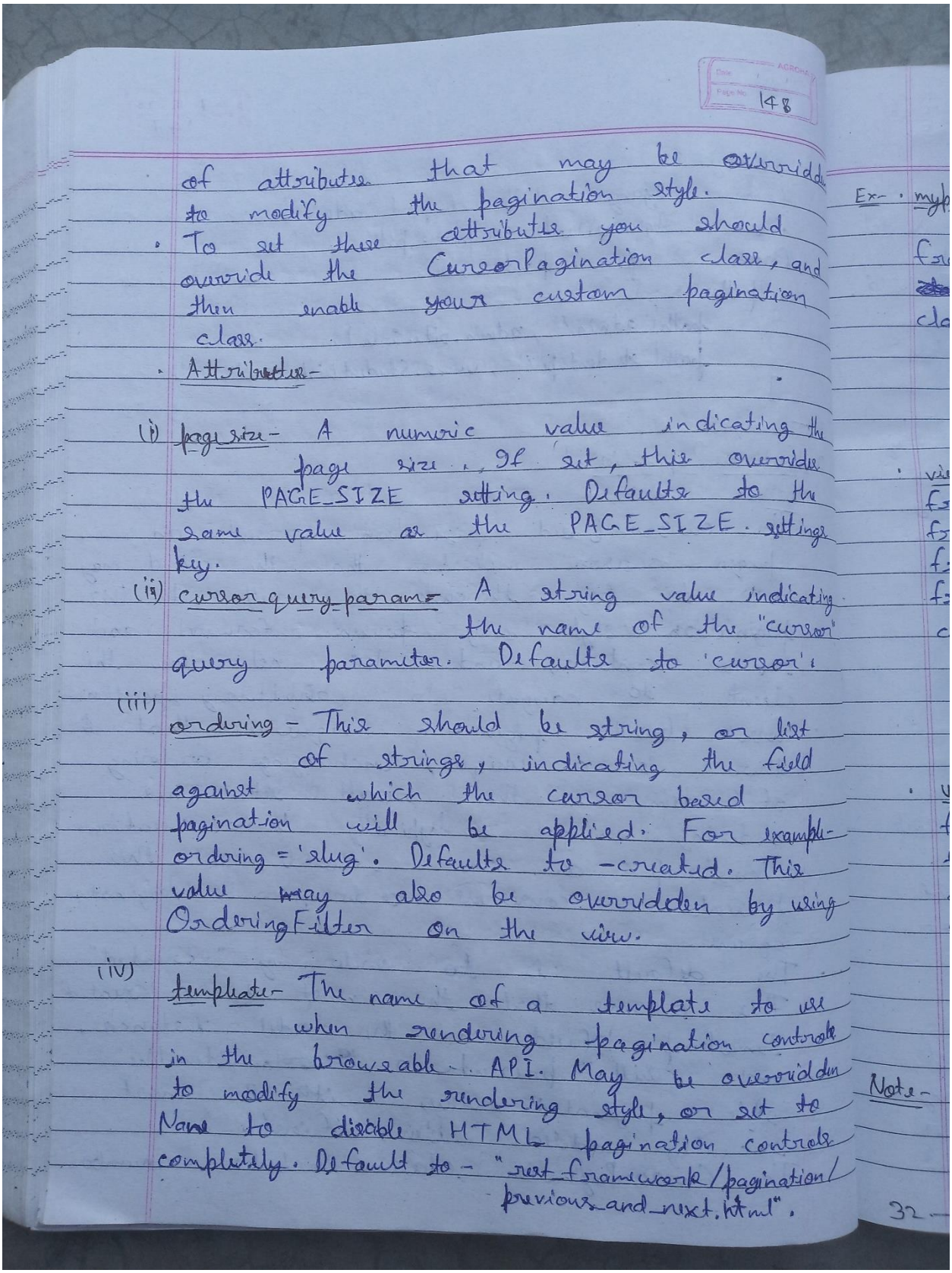
- urls.py -

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/', views.StudentList.as_view()),
]
```

31 —

3- Cursor Pagination -

- The cursor-based pagination presents an opaque "cursor" indicator that the client may use to page through the result set. This pagination style only presents forward and reverse controls, and does not allow the client to navigate to arbitrary positions.
- Cursor based pagination requires that there is a unique, unchanging ordering of items in the result set. This ordering might typically be a creation timestamp on the records, as this presents a consistent ordering to paginate against.
- The default is to order by "-created". This assumes that there must be a 'created' timestamp field on the model instance, and will present a ~~"creation~~ "timeline" style paginated view, with the most recently added items first.
- The CursorPagination class includes a number



of attributes that may be overridden to modify the pagination style.

- To set these attributes you should override the CursorPagination class, and then enable your custom pagination class.
- Attributes -

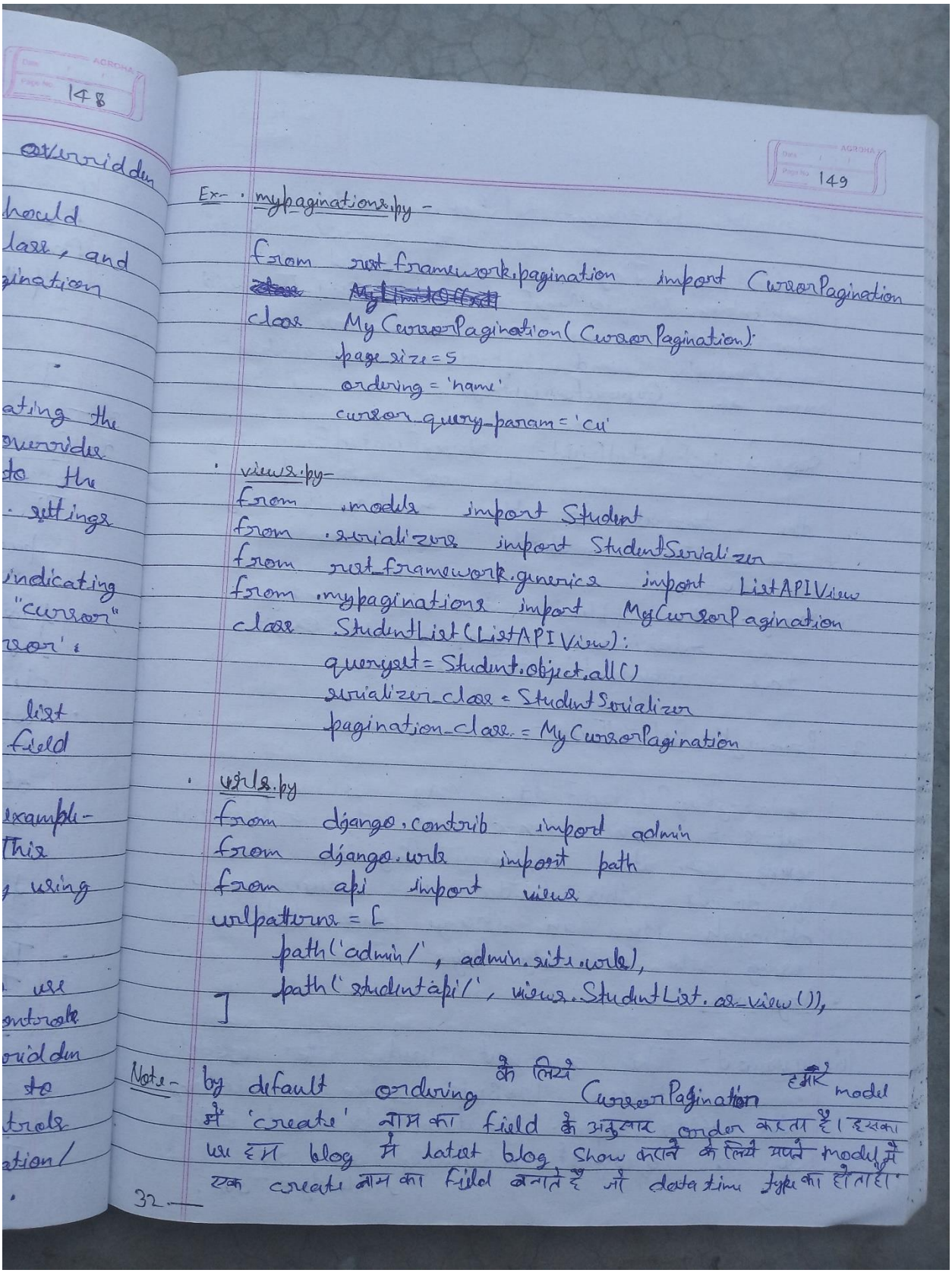
(i) page_size - A numeric value indicating the page size. If set, this overrides the PAGE_SIZE setting. Defaults to the same value as the PAGE_SIZE settings key.

(ii) cursor_query_param = A string value indicating the name of the "cursor" query parameter. Defaults to 'cursor'.

(iii) ordering - This should be string, or list of strings, indicating the field against which the cursor based pagination will be applied. For example- ordering = 'slug'. Defaults to -created. This value may also be overridden by using OrderingFilter on the view.

(iv) template - The name of a template to use when rendering pagination controls in the browseable API. May be overridden to modify the rendering style, or set to None to disable HTML pagination controls completely. Default to - "rest_framework/pagination/previous_and_next.html".

Note -



148

overridden
should
class, and
gination

ating the
overrides
to the
settings

indicating
"cursor"
sor':

list
field

example-
This
using

use
ontrols
ridden
the
trols
ation/

32—

149

Ex- · mypaginations.py -

```
from rest_framework.pagination import CursorPagination
class MyCursorPagination(CursorPagination):
    page_size = 5
    ordering = 'name'
    cursor_query_param = 'cu'
```

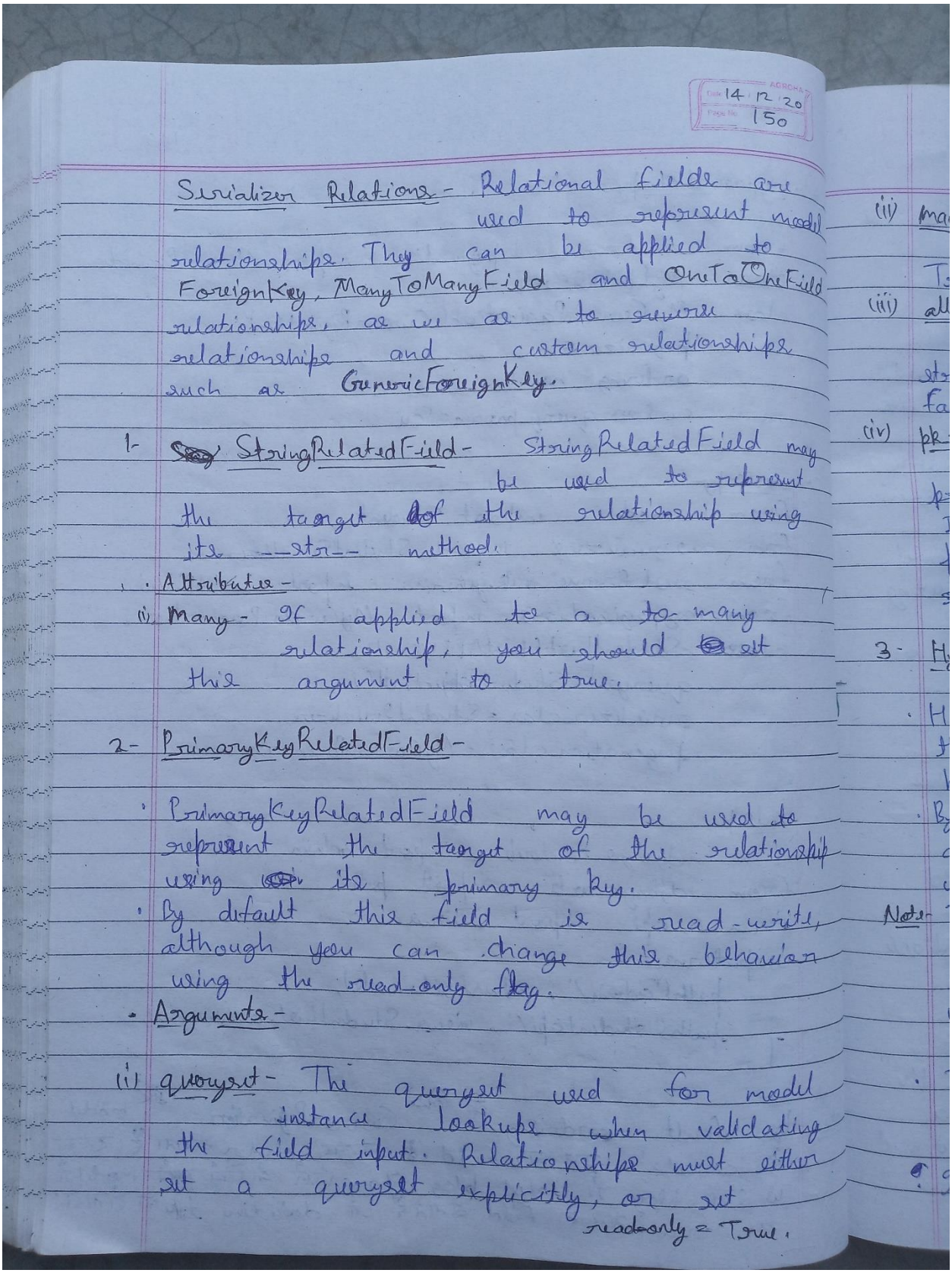
· views.py-

```
from .models import Student
from .serializers import StudentSerializer
from rest_framework.generics import ListAPIView
from .mypaginations import MyCursorPagination
class StudentList(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    pagination_class = MyCursorPagination
```

· urls.py

```
from django.contrib import admin
from django.urls import path
from api import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('studentapi/', views.StudentList.as_view()),
]
```

Note- by default ordering के लिए CursorPagination हमारे model में 'create' नाम का field के अनुसार order करता है। इसका use हम blog में latest blog show कराने के लिए अपने model में एक create नाम का field बनाते है जो datetime type का होता है।



Date 14/12/20
Page No. 150

Serializer Relations - Relational fields are used to represent model relationships. They can be applied to ForeignKey, ManyToManyField and OneToOneField relationships, as well as to reverse relationships and custom relationships such as GenericForeignKey.

1- StringRelatedField - StringRelatedField may be used to represent the target of the relationship using its __str__ method.

· Attributes -

(i) many - If applied to a to-many relationship, you should set this argument to true.

2- PrimaryKeyRelatedField -

· PrimaryKeyRelatedField may be used to represent the target of the relationship using its primary key.

· By default this field is read-write, although you can change this behavior using the read_only flag.

· Arguments -

(i) queryset - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set read_only = True.

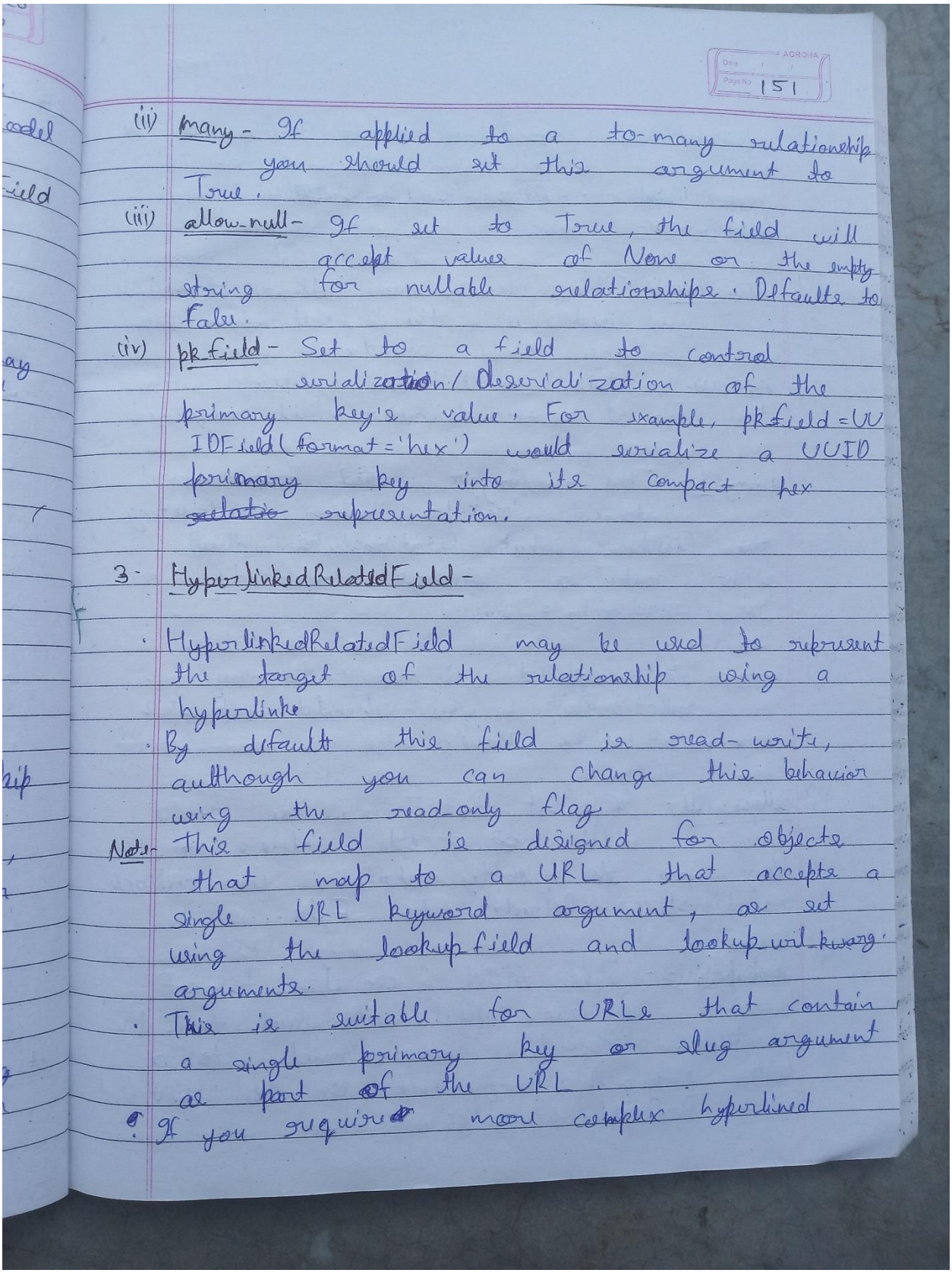
(ii) ma

(iii) all

(iv) pk

3- H

Note-



(ii) many - If applied to a to-many relationship you should set this argument to True.

(iii) allow_null - If set to True, the field will accept values of None or the empty string for nullable relationships. Defaults to false.

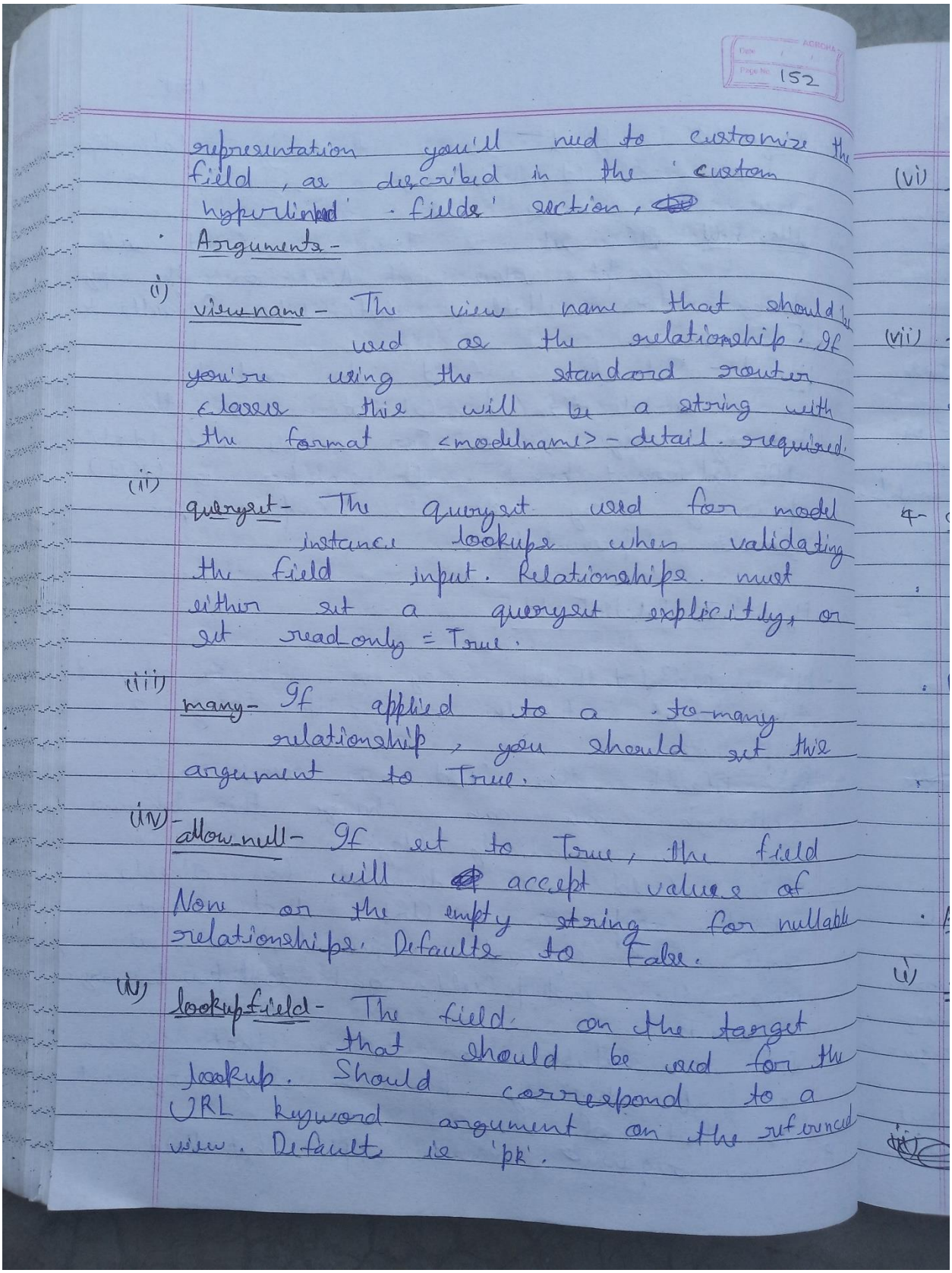
(iv) pk_field - Set to a field to control serialization/deserialization of the primary key's value. For example, pk_field=UUIDField(format='hex') would serialize a UUID primary key into its compact hex representation.

3- HyperlinkedRelatedField -

- HyperlinkedRelatedField may be used to represent the target of the relationship using a hyperlink
- By default this field is read-write, although you can change this behavior using the read_only flag

Note- This field is designed for objects that map to a URL that accepts a single URL keyword argument, as set using the lookup_field and lookup_url_kwarg arguments.

- This is suitable for URLs that contain a single primary key or slug argument as part of the URL.
- If you require more complex hyperlinked



representation you'll need to customize the field, as described in the 'custom hyperlinked fields' section.

Arguments -

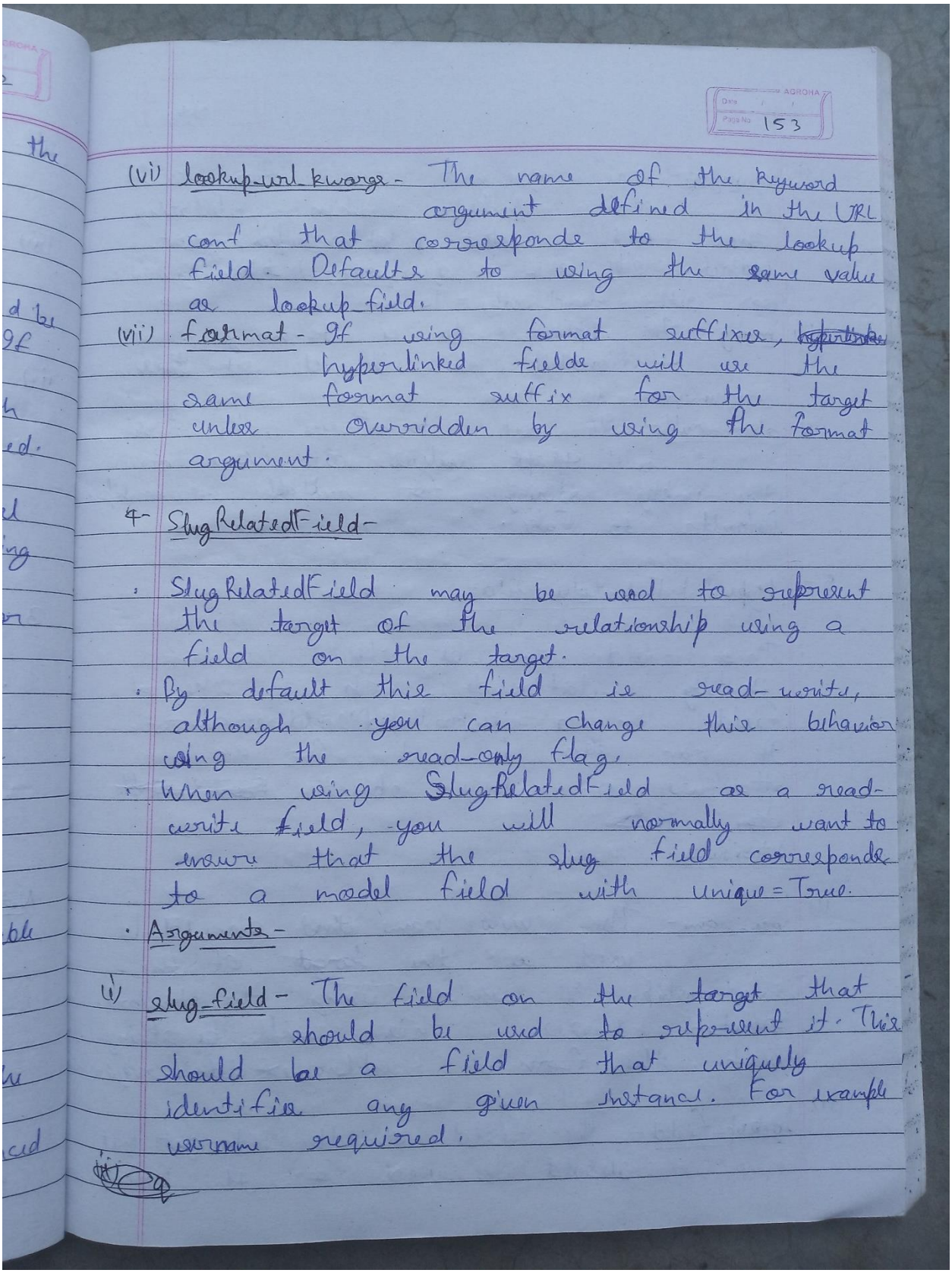
(i) view_name - The view name that should be used as the relationship. If you're using the standard router classes this will be a string with the format <modelname>-detail. required.

(ii) queryset - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set read_only = True.

(iii) many - If applied to a to-many relationship, you should set this argument to True.

(iv) allow_null - If set to True, the field will accept values of None or the empty string for nullable relationships. Defaults to False.

(v) lookup_field - The field on the target that should be used for the lookup. Should correspond to a URL keyword argument on the referenced view. Defaults to 'pk'.



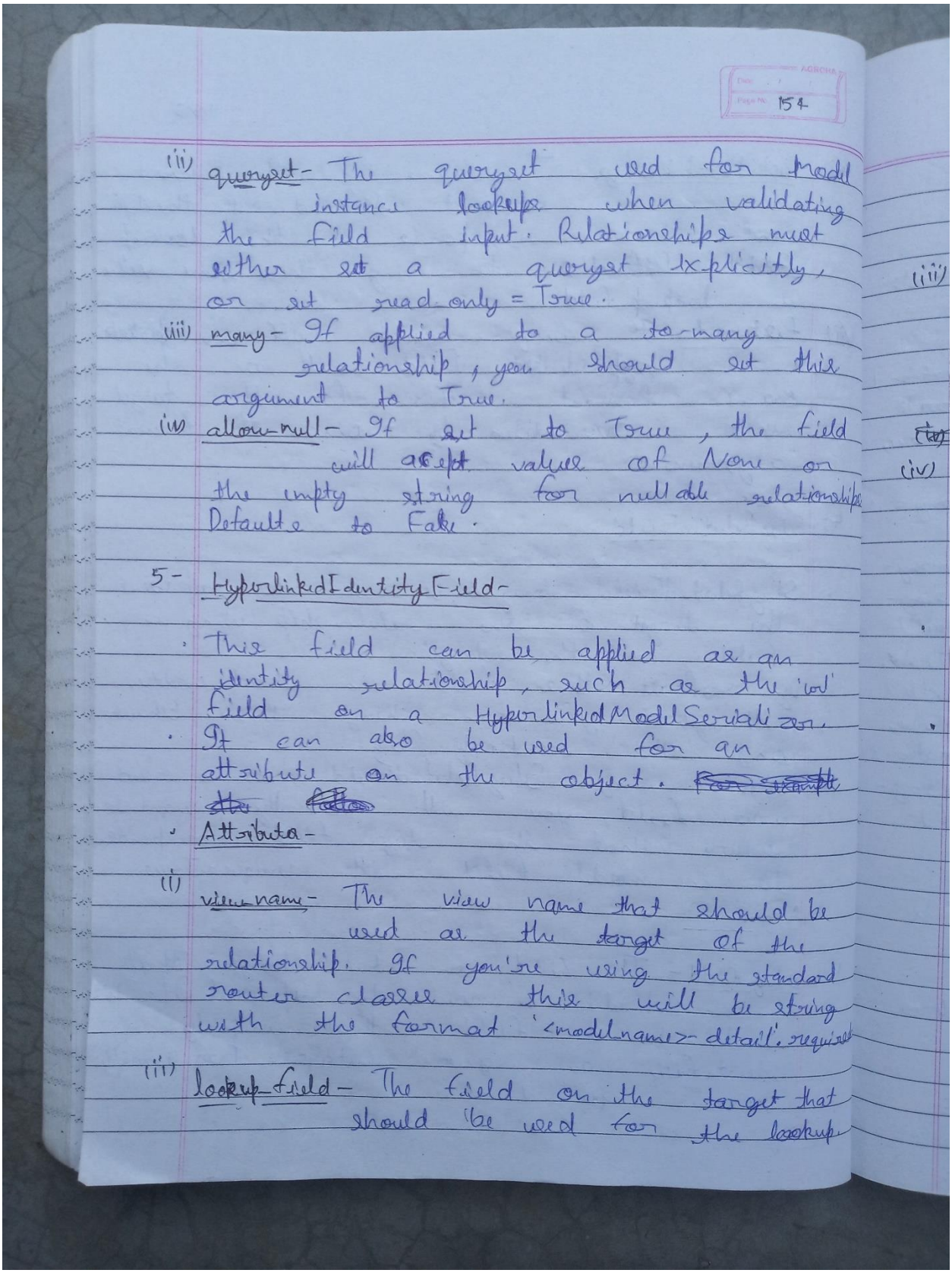
(vi) lookup_url_kwarge - The name of the keyword argument defined in the URL conf that corresponds to the lookup field. Defaults to using the same value as lookup_field.

(vii) format - If using format suffixes, hyperlinked fields will use the same format suffix for the target unless overridden by using the format argument.

4- SlugRelatedField -

- SlugRelatedField may be used to represent the target of the relationship using a field on the target.
- By default this field is read-write, although you can change this behavior using the read_only flag.
- When using SlugRelatedField as a read-write field, you will normally want to ensure that the slug field corresponds to a model field with unique=True.
- Arguments -

(i) slug_field - The field on the target that should be used to represent it. This should be a field that uniquely identifies any given instance. For example username required.



(ii) queryset - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set read_only = True.

(iii) many - If applied to a to-many relationship, you should set this argument to True.

(iv) allow_null - If set to True, the field will accept values of None or the empty string for nullable relationships. Defaults to False.

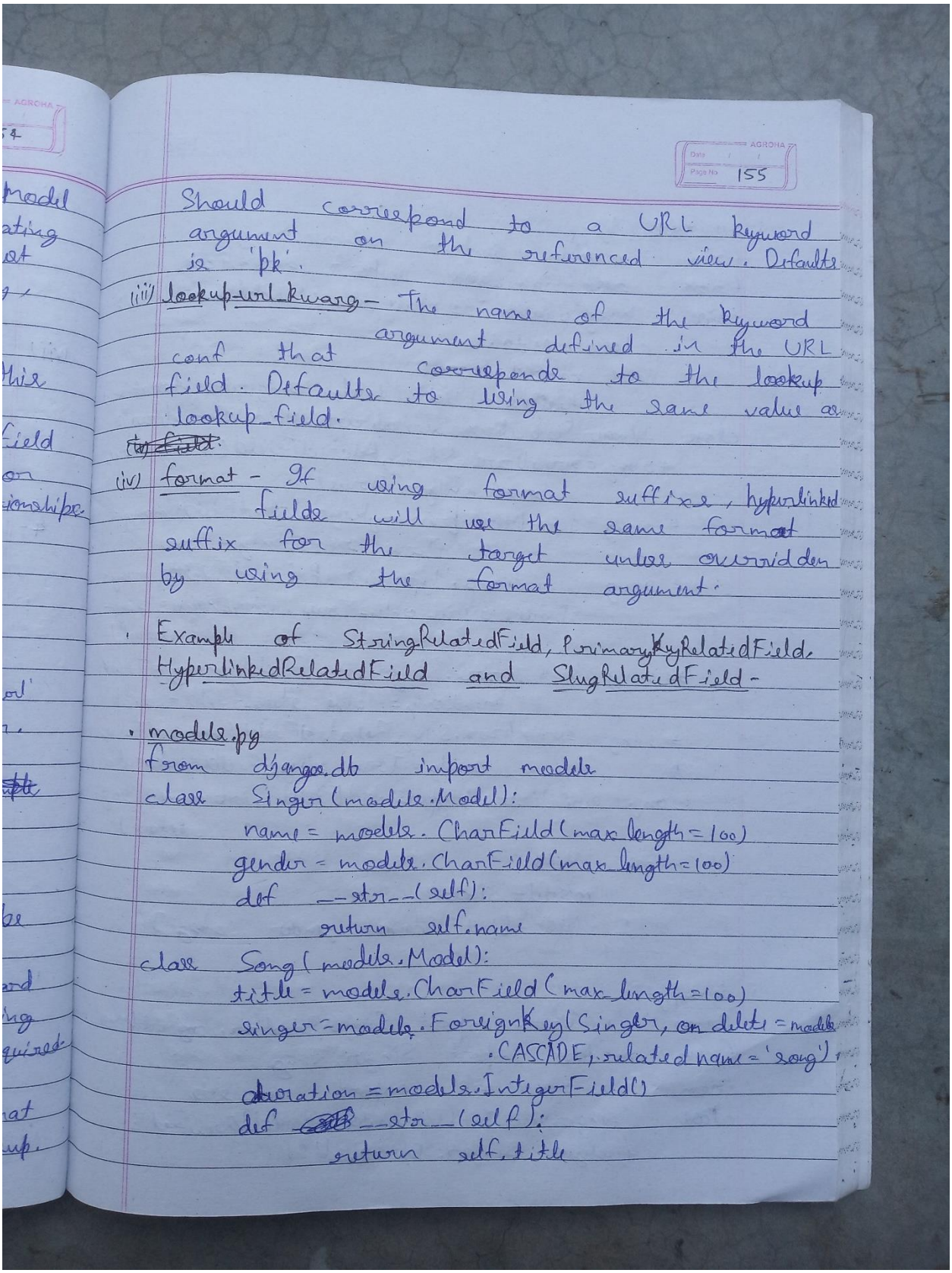
5 - HyperlinkedIdentityField -

- This field can be applied as an identity relationship, such as the 'url' field on a HyperlinkedModelSerializer.
- It can also be used for an attribute on the object.

- Attributes -

(i) view_name - The view name that should be used as the target of the relationship. If you're using the standard router classes this will be string with the format '<model_name>-detail'. required

(ii) lookup_field - The field on the target that should be used for the lookup.



Should correspond to a URL keyword argument on the referenced view. Default is 'pk'.

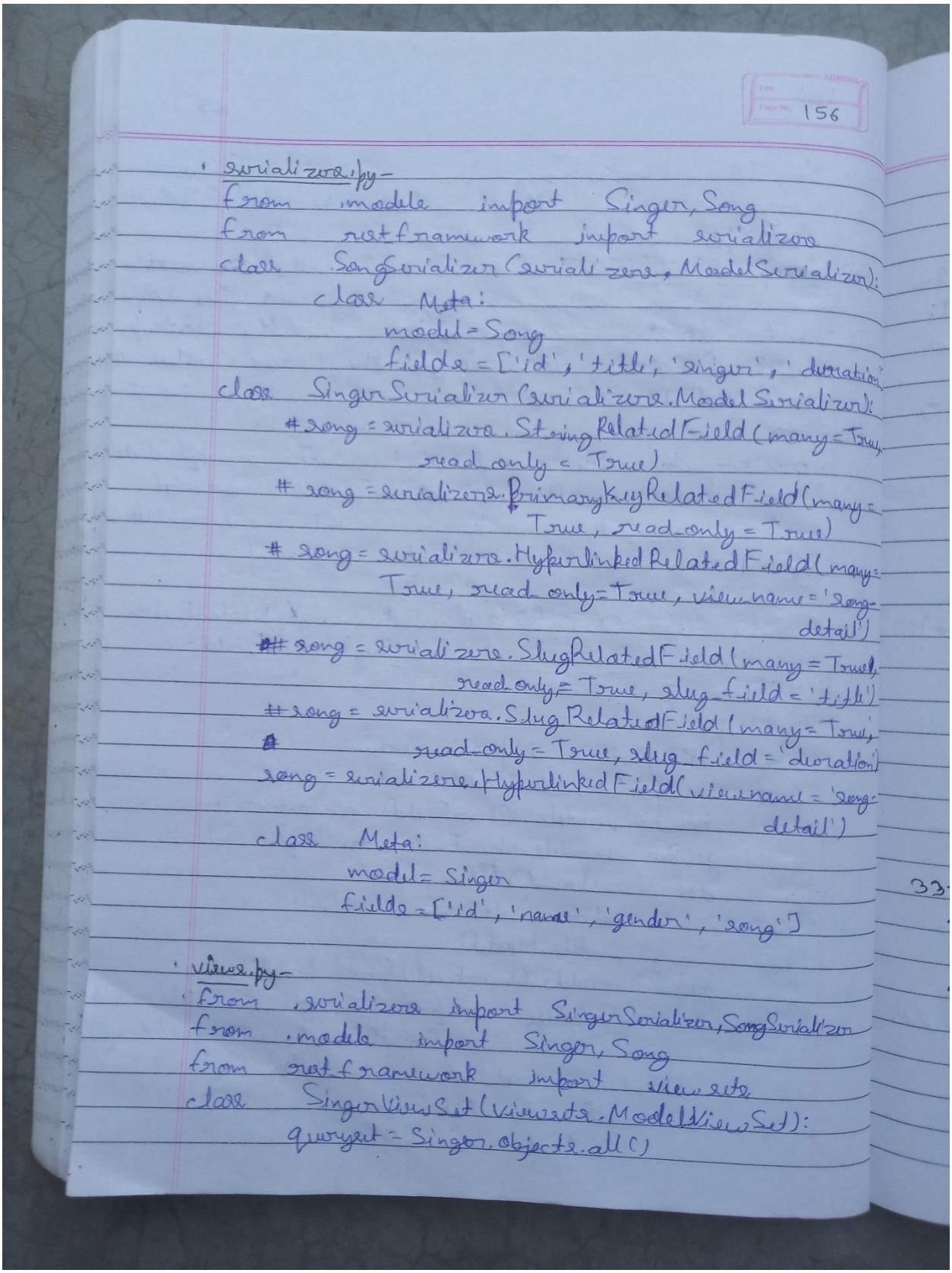
(iii) lookup_url_kwarg - The name of the keyword argument defined in the URL conf that corresponds to the lookup field. Defaults to using the same value as lookup_field.

(iv) format - If using format suffixes, hyperlinked fields will use the same format suffix for the target unless overridden by using the format argument.

- Example of StringRelatedField, PrimaryKeyRelatedField, HyperlinkedRelatedField and SlugRelatedField -

- models.py

```
from django.db import models
class Singer(models.Model):
    name = models.CharField(max_length=100)
    gender = models.CharField(max_length=100)
    def __str__(self):
        return self.name
class Song(models.Model):
    title = models.CharField(max_length=100)
    singer = models.ForeignKey(Singer, on_delete=models
                .CASCADE, related_name='song')
    duration = models.IntegerField()
    def __str__(self):
        return self.title
```



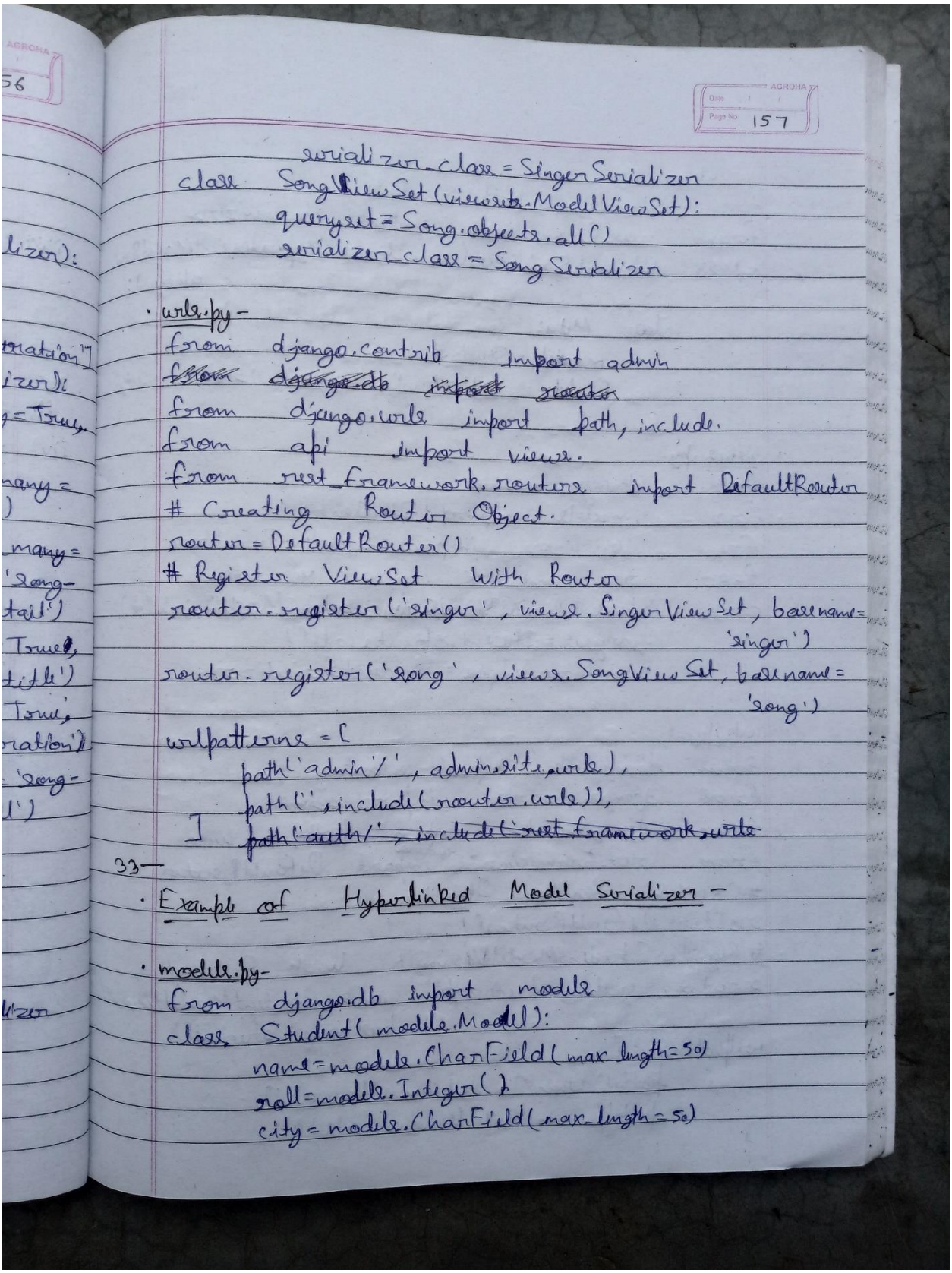
156

serializers.py -

```
from .models import Singer, Song
from rest_framework import serializers
class SongSerializer(serializers.ModelSerializer):
    class Meta:
        model = Song
        fields = ['id', 'title', 'singer', 'duration']
class SingerSerializer(serializers.ModelSerializer):
    #song = serializers.StringRelatedField(many=True,
                read_only = True)
    # song = serializers.PrimaryKeyRelatedField(many=
                True, read_only = True)
    # song = serializers.HyperlinkedRelatedField(many=
                True, read_only=True, view_name='song-
                detail')
    ## song = serializers.SlugRelatedField(many=True,
                read_only=True, slug_field='title')
    #song = serializers.SlugRelatedField(many=True,
    #            read_only=True, slug_field='duration')
    song = serializers.HyperlinkedField(view_name='song-
                detail')
    class Meta:
        model= Singer
        fields=['id', 'name', 'gender', 'song']
```

views.py -

```
from .serializers import SingerSerializer, SongSerializer
from .models import Singer, Song
from rest_framework import viewsets
class SingerViewSet(viewsets.ModelViewSet):
    queryset = Singer.objects.all()
```



```
    serializer_class = SingerSerializer
class SongViewSet(viewsets.ModelViewSet):
    queryset = Song.objects.all()
    serializer_class = SongSerializer
```

· urls.py -

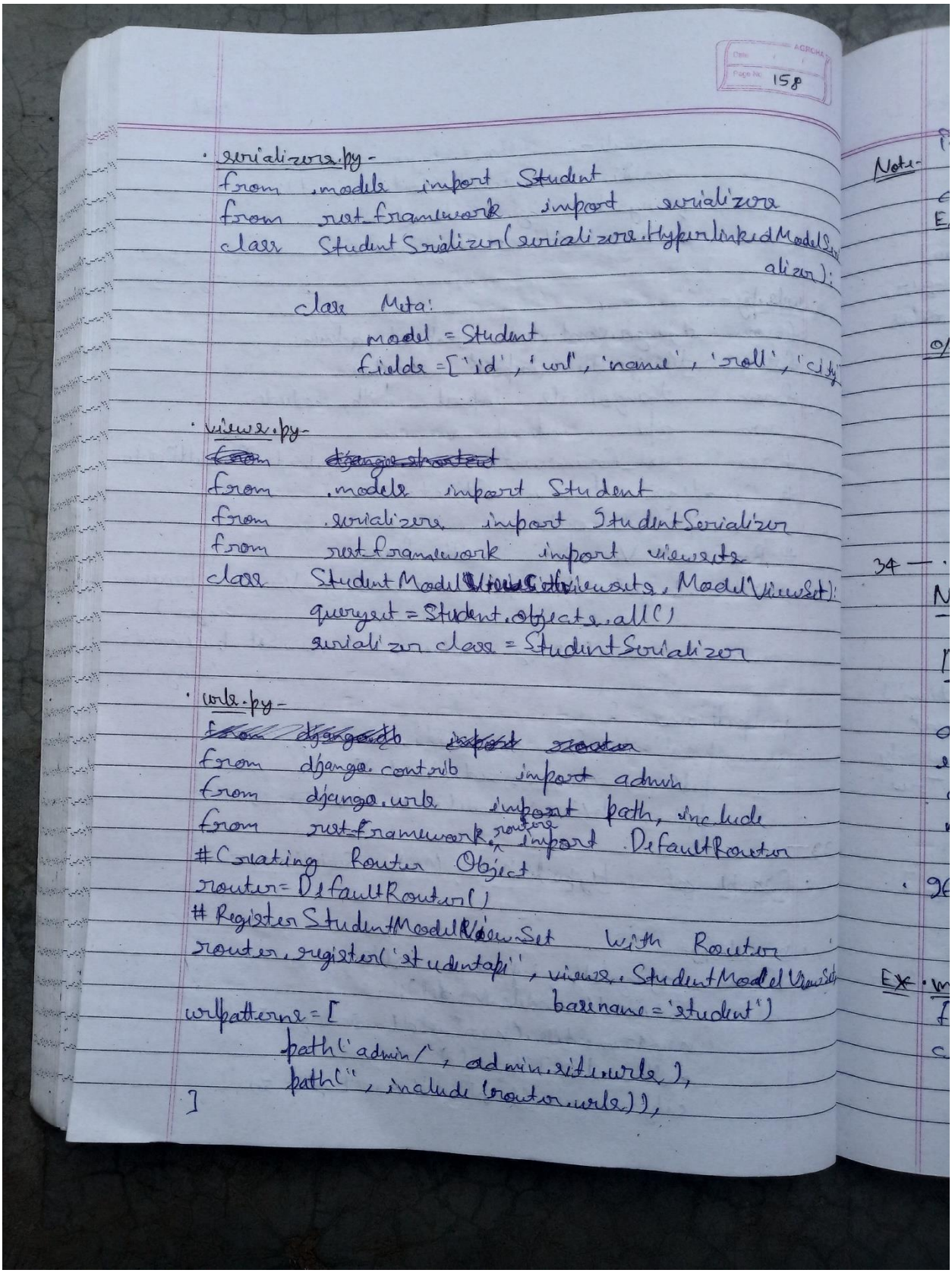
```
from django.contrib import admin
from django.db import router
from django.urls import path, include.
from api import views.
from rest_framework.routers import DefaultRouter
# Creating Router Object.
router = DefaultRouter()
# Register ViewSet With Router
router.register('singer', views.SingerViewSet, basename='singer')
router.register('song', views.SongViewSet, basename='song')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
    path('auth/', include('rest_framework.urls'
]
```

33—

· Example of Hyperlinked Model Serializer -

· models.py-

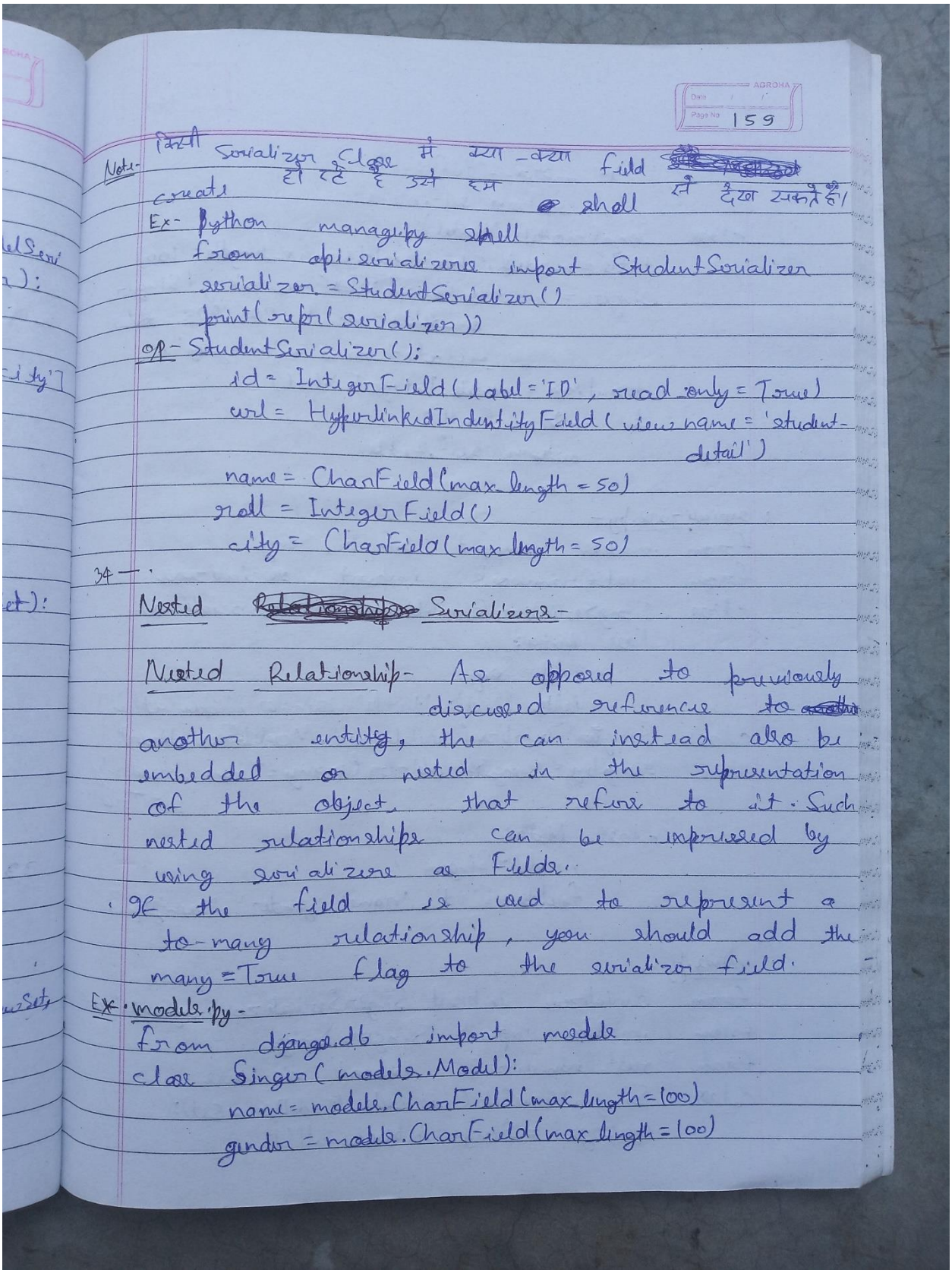
```
from django.db import models
class Student(models.Model):
    name = models.CharField(max_length=50)
    roll = models.Integer()
    city = models.CharField(max_length=50)
```



· serializers.py -
from .models import Student
from rest_framework import serializers
class StudentSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'url', 'name', 'roll', 'city']

· views.py -
from .models import Student
from .serializers import StudentSerializer
from rest_framework import viewsets
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer

· urls.py -
from django.contrib import admin
from django.urls import path, include
from rest_framework.routers import DefaultRouter
#Creating Router Object
router = DefaultRouter()
#Register StudentModelViewSet with Router
router.register('studentapi', views.StudentModelViewSet, basename='student')
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
]



Page No 159

Note- जितनी Serializer Class में क्या-क्या field हो रहे है उसे हम create कर सकते है shell में देख सकते है।

Ex- Python manage.py shell
from api.serializers import StudentSerializer
serializer = StudentSerializer()
print(repr(serializer))

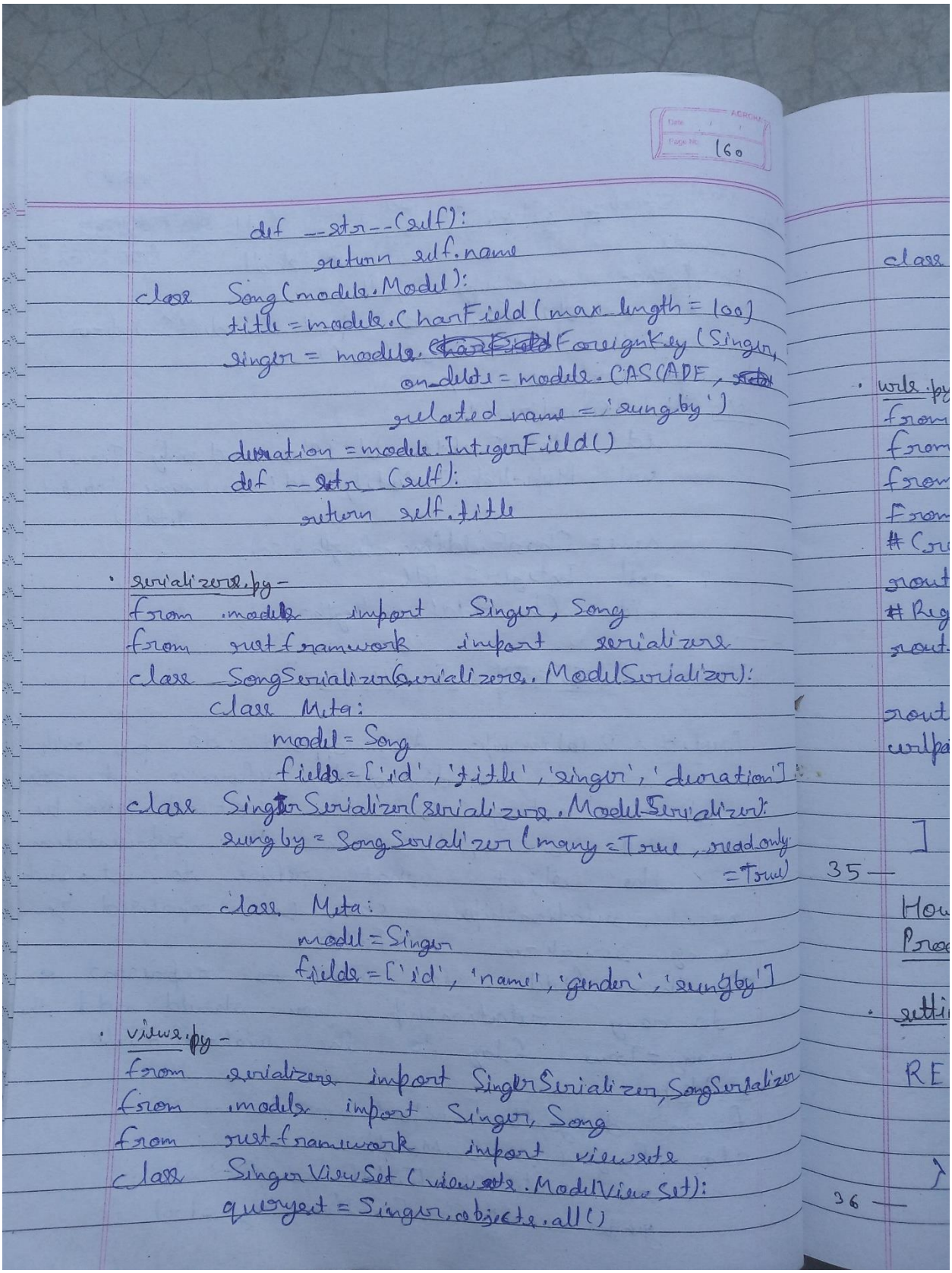
Op- StudentSerializer():
id = IntegerField(label='ID', read_only=True)
url = HyperlinkedIdentityField(view_name='student-detail')
name = CharField(max_length=50)
roll = IntegerField()
city = CharField(max_length=50)

34 —

Nested ~~Relationships~~ Serializers-

Nested Relationship- As opposed to previously discussed reference to another entity, the can instead also be embedded or nested in the representation of the object that refers to it. Such nested relationships can be expressed by using serializers as Fields.
If the field is used to represent a to-many relationship, you should add the many=True flag to the serializer field.

Ex. models.py-
from django.db import models
class Singer(models.Model):
name = models.CharField(max_length=100)
gender = models.CharField(max_length=100)



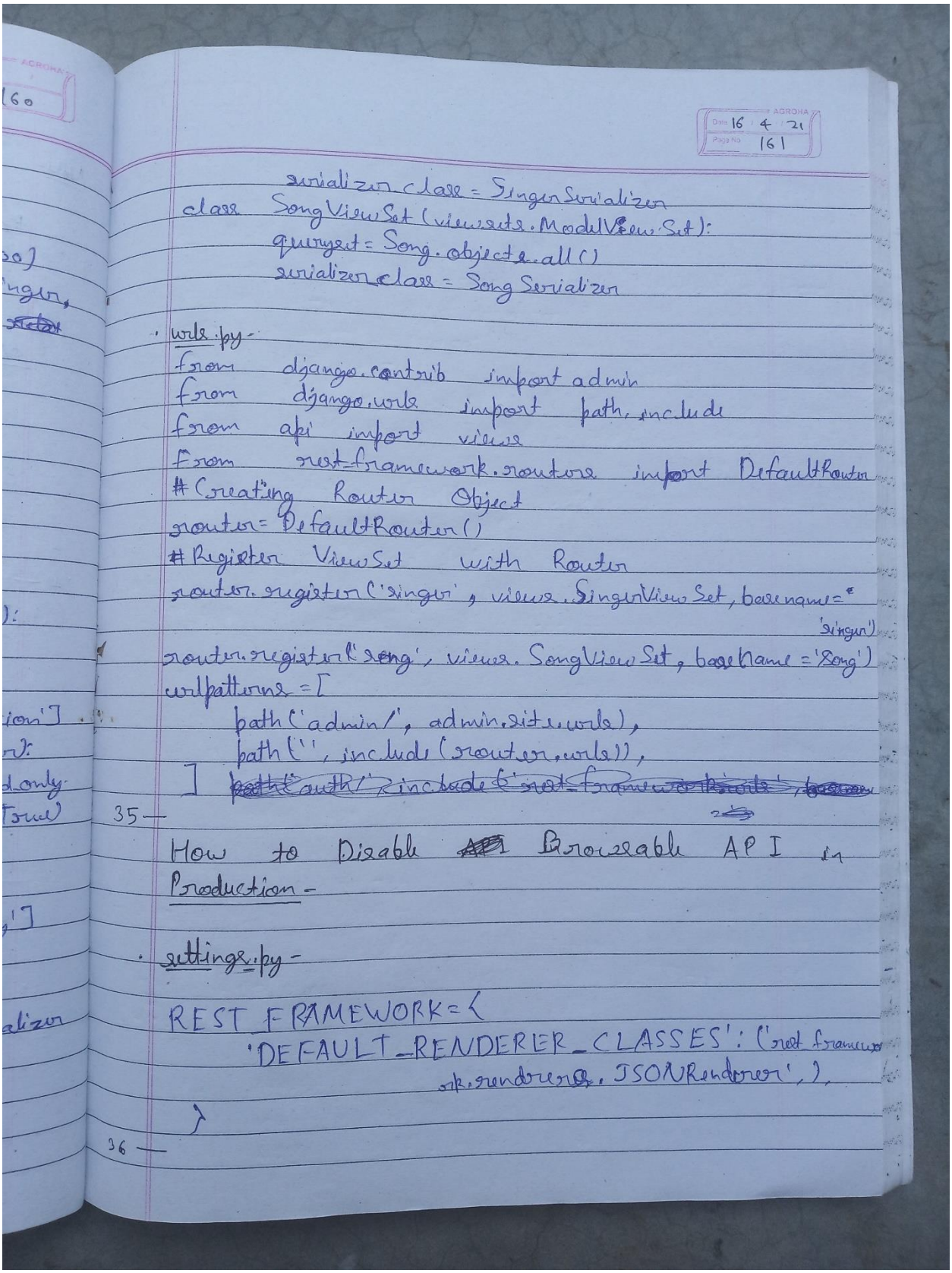
```
    def __str__(self):
        return self.name
class Song(models.Model):
    title = models.CharField(max_length=100)
    singer = models.ForeignKey(Singer,
                on_delete=models.CASCADE,
                related_name='sungby')
    duration = models.IntegerField()
    def __str__(self):
        return self.title
```

- serializers.py -

```
from .models import Singer, Song
from rest_framework import serializers
class SongSerializer(serializers.ModelSerializer):
    class Meta:
        model = Song
        fields = ['id', 'title', 'singer', 'duration']
class SingerSerializer(serializers.ModelSerializer):
    sungby = SongSerializer(many=True, read_only
                =True)
    class Meta:
        model = Singer
        fields = ['id', 'name', 'gender', 'sungby']
```

- views.py -

```
from serializers import SingerSerializer, SongSerializer
from .models import Singer, Song
from rest_framework import viewsets
class SingerViewSet(viewsets.ModelViewSet):
    queryset = Singer.objects.all()
```



```
    serializer_class = SingerSerializer
class SongViewSet(viewsets.ModelViewSet):
    queryset = Song.objects.all()
    serializer_class = SongSerializer
```

- urls.py -

```
from django.contrib import admin
from django.urls import path, include
from api import views
From rest_framework.routers import DefaultRouter
# Creating Router Object
router = DefaultRouter()
# Register ViewSet with Router
router.register('singer', views.SingerViewSet, basename='singer')
router.register('song', views.SongViewSet, basename='song')
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls)),
]
```

35 —

How to Disable Browsable API in Production -

- settings.py -

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': ('rest_framework.renderers.JSONRenderer',),
}
```

36 —