Django Channels

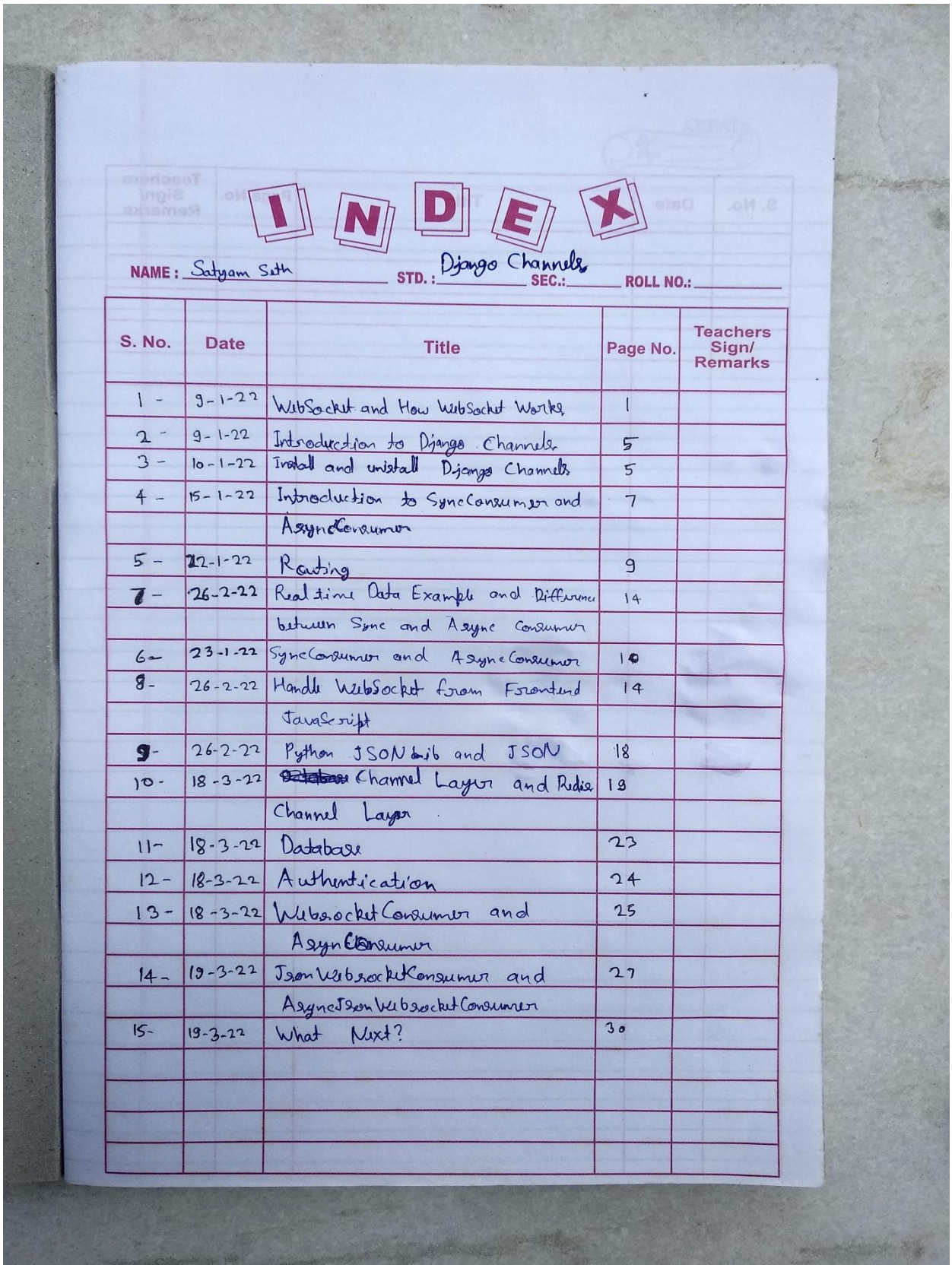
GEEKYSHOWS YOUTUBE CHANNEL LEARNING NOTES

Source Code - https://github.com/satyam-seth-learnings/django_channels_learning/tree/master/Geekyshows

YouTube Link - https://youtube.com/playlist?list=PLbGui_ZYuhij6LpUbWgKUxggL_AuoHHVw

SATYAM SETH

04-09-2022



INDEX

NAME : Satyam Seth STD. : Django Channels SEC.: ROLL NO.:

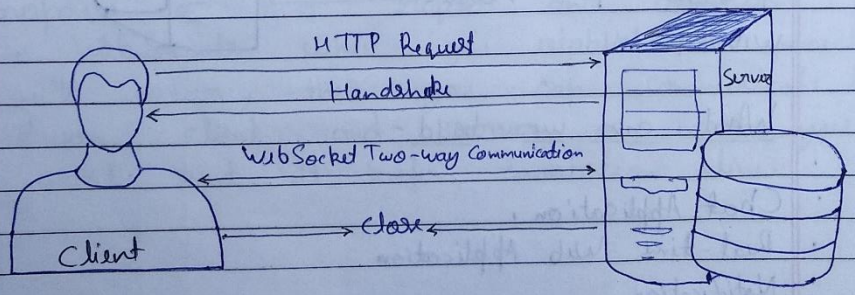
ARFAT
Date 9-1-22
Page 1

WebSocket -

- WebSocket is a full-duplex (two-way communication) protocol that is used in the same scenario of client-server communication.
- It is a stateful protocol, which means the connection between client and server will keep alive until it is terminated by either client or server, after closing the connection by either of the client and server, the connection is terminated from both the end.
- WebSockets do not use the http:// or https:// scheme (because they do not follow the HTTP protocol).
- Rather, WebSocket URIs use a new scheme ws: (or wss: for a secure WebSocket).
- The remainder of the URI is the same as an HTTP URI: a host, port, path and any query parameters.

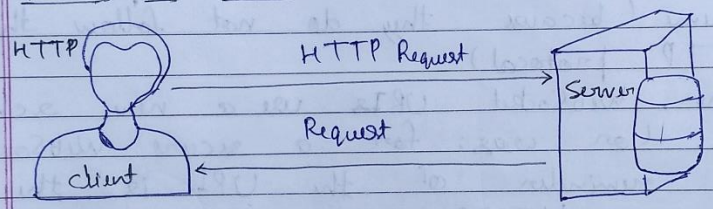
Example - ws://example.com:8000/ws/chat

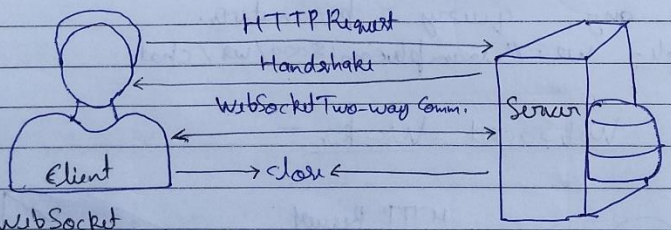
How WebSocket Works -



- Client sends regular HTTP request with an additional header to be requested
- The Server gets the HTTP request and notices the request for the Upgrade header. This lets the Server know that we are requesting for a WebSocket connection.
- If all goes well persistent connection established between client and server.
- Connection can be closed either by client or server.

HTTP vs WebSocket -

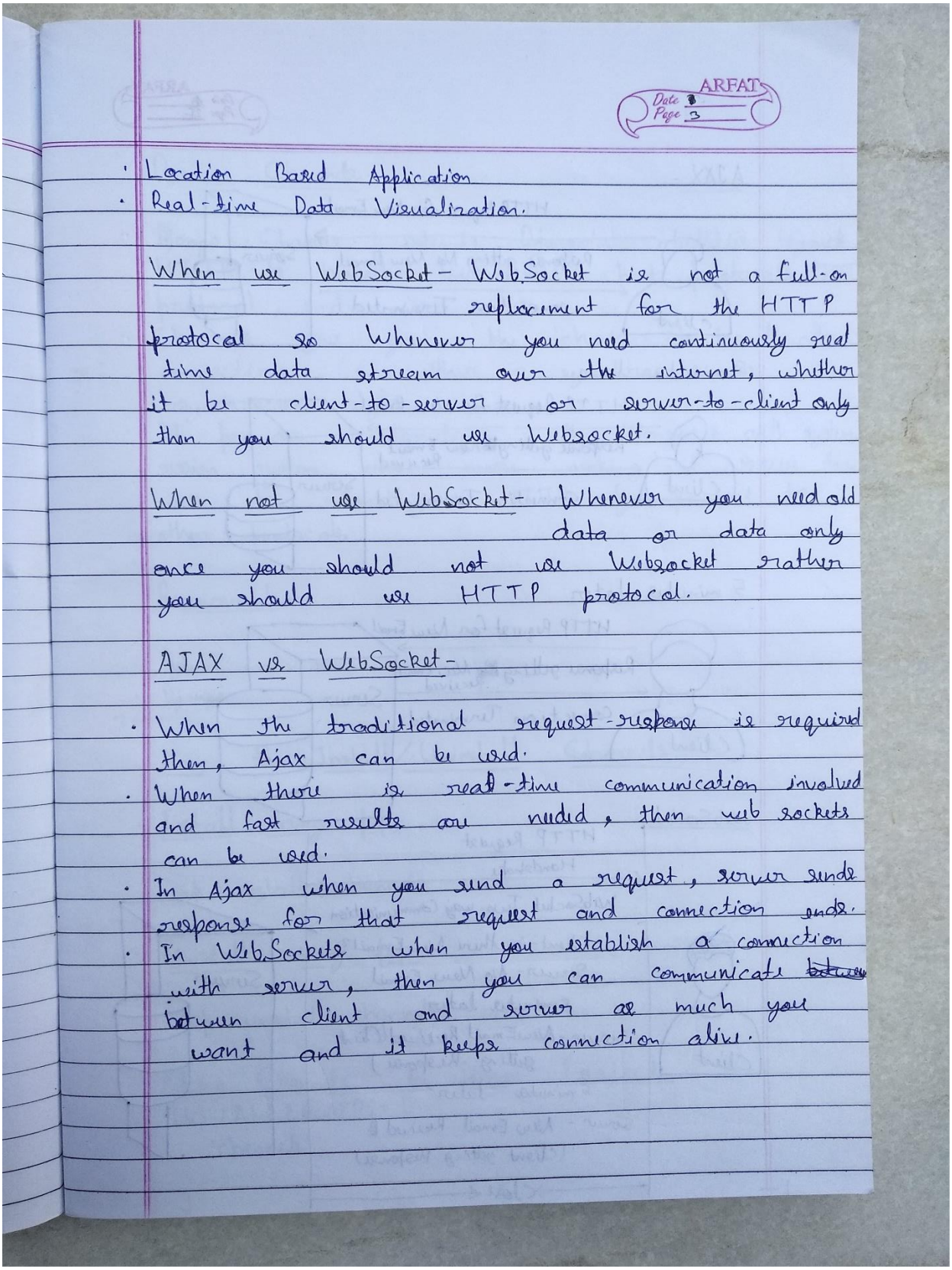




WebSocket

What can we build -

- Chat Application,
- Real-time Web Application
- Notification
- Trading



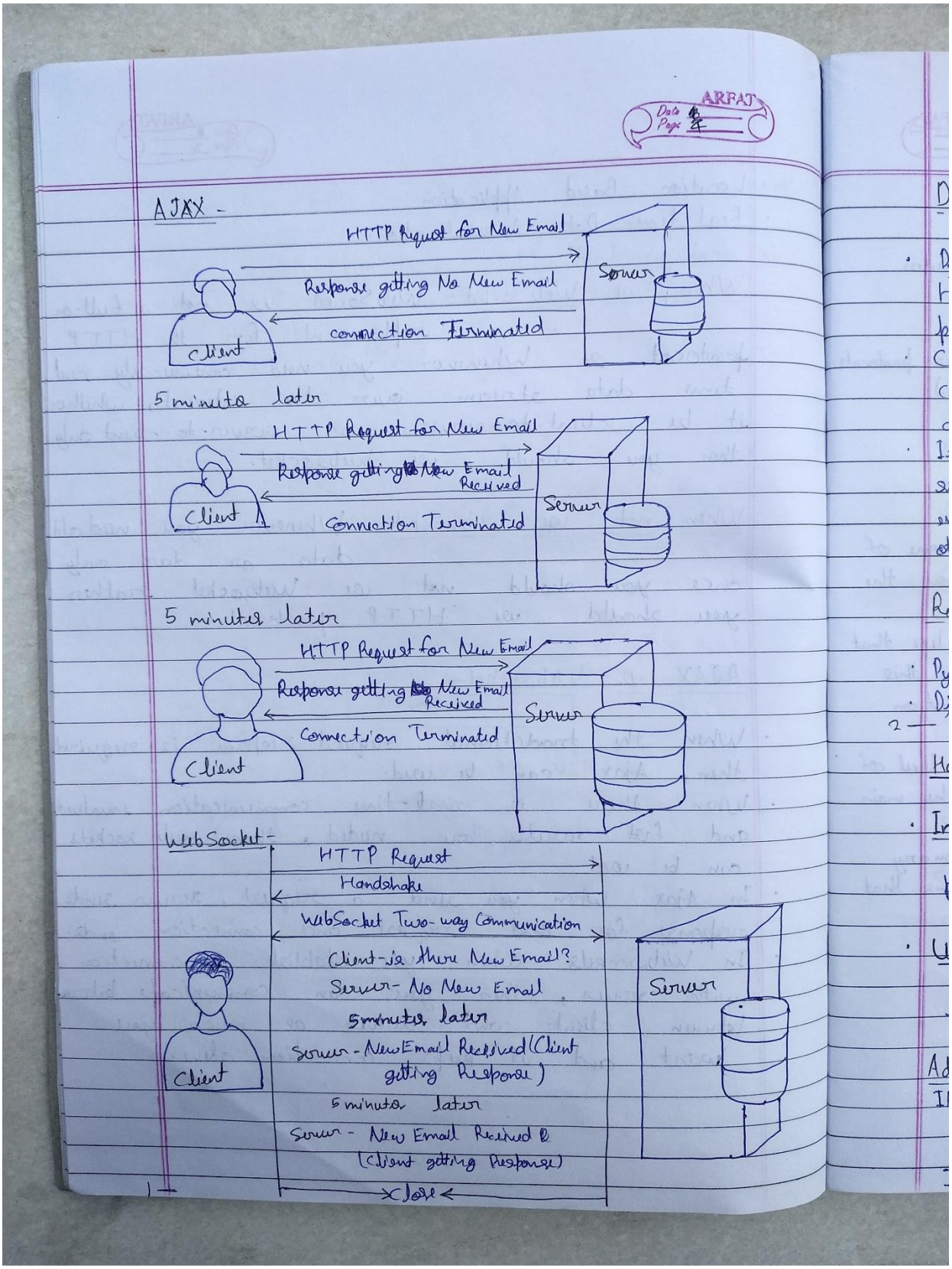
- Location Based Application
- Real-time Data Visualization.

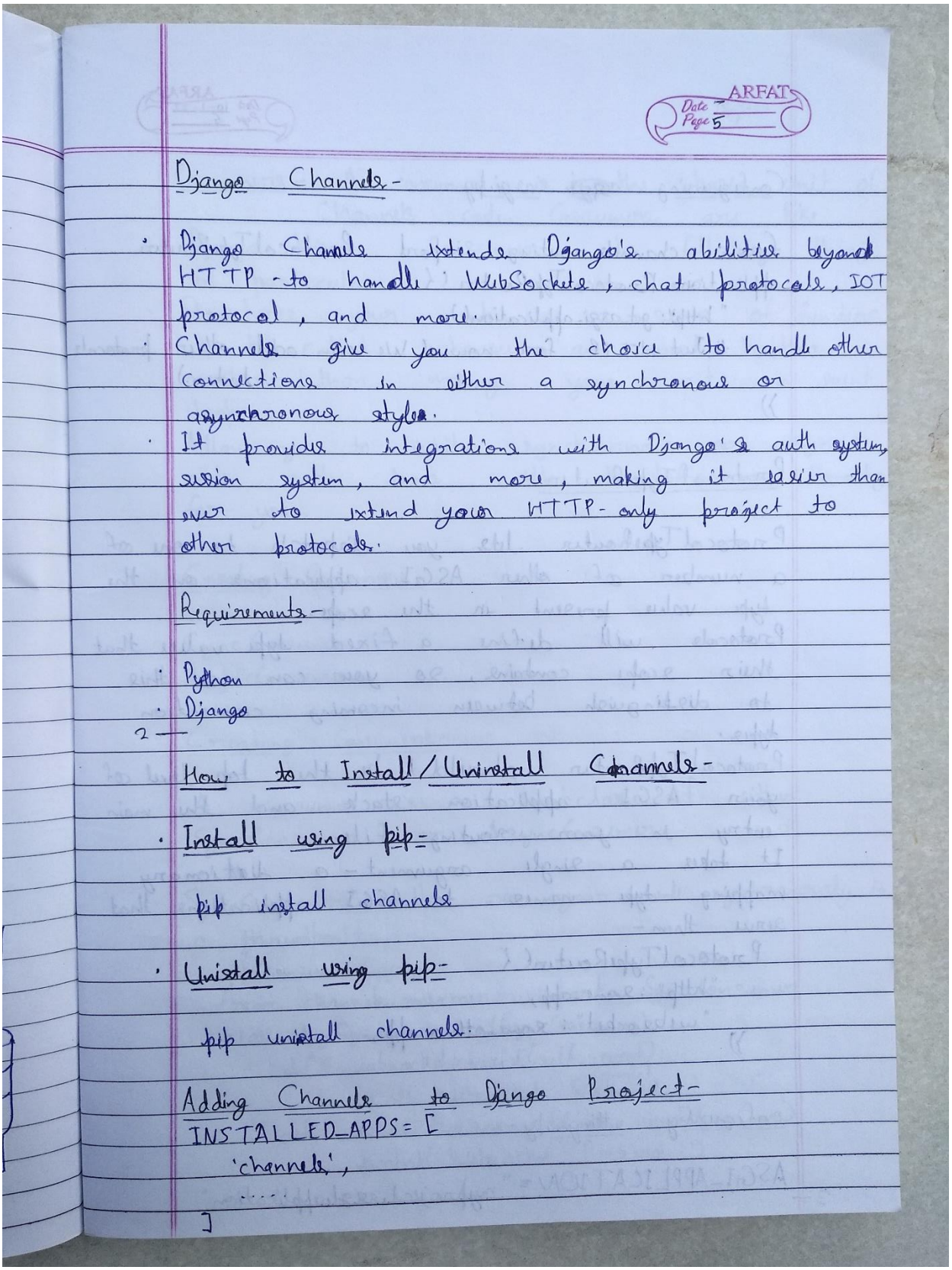
When use WebSocket - WebSocket is not a full-on replacement for the HTTP protocal so Whenever you need continuously real time data stream over the internet, whether it be client-to-server or server-to-client only then you should use Websocket.

When not use WebSocket - Whenever you need old data or data only once you should not use Websocket rather you should use HTTP protocal.

AJAX vs WebSocket -

- When the traditional request-response is required then, Ajax can be used.
- When there is real-time communication involved and fast results are needed, then web sockets can be used.
- In Ajax when you send a request, server sends response for that request and connection ends.
- In WebSockets when you establish a connection with server, then you can communicate between client and server as much you want and it keeps connection alive.


ARPAT
AJAX -
HTTP Request for New Email
Response getting No New Email
connection Terminated
Client
Server
5 minutes later
HTTP Request for New Email
Response getting New Email Received
Connection Terminated
Client
Server
5 minutes later
HTTP Request for New Email
Response getting New Email Received
Connection Terminated
Client
Server
WebSocket -
HTTP Request
Handshake
WebSocket Two-way Communication
Client - is there New Email?
Server - No New Email
5 minutes later
Server - New Email Received (Client getting Response)
5 minutes later
Server - New Email Received (Client getting Response)
Close
Client
Server



ARFAT
Date
Page 5

Django Channels -

- Django Channels extends Django's abilities beyond HTTP - to handle WebSockets, chat protocols, IOT protocol, and more.
- Channels give you the choice to handle other connections in either a synchronous or asynchronous style.
- It provides integrations with Django's auth system, session system, and more, making it easier than ever to extend your HTTP-only project to other protocols.

Requirements -

- Python
- Django

2 —

How to Install / Uninstall Channels -

- **Install using pip -**

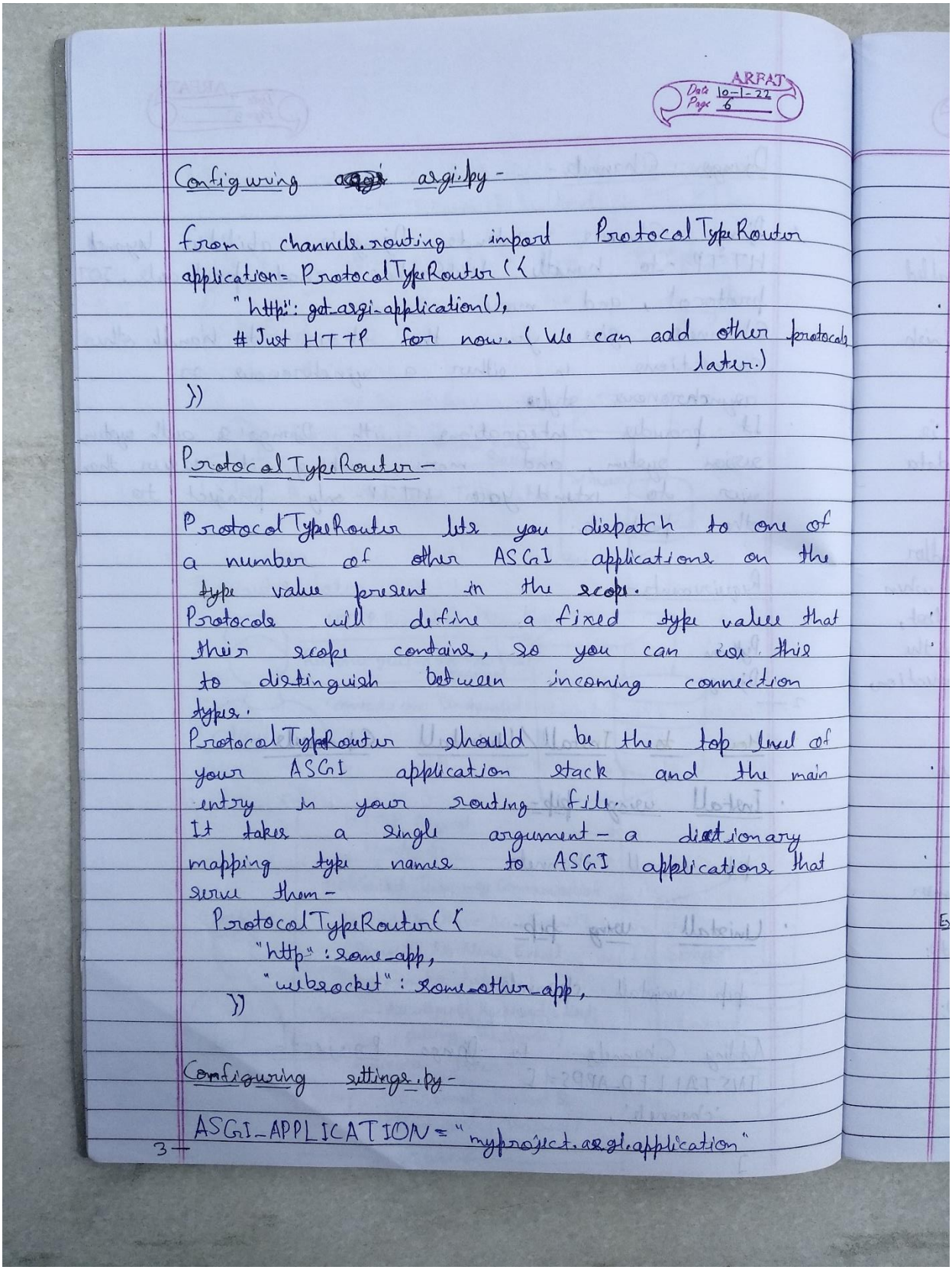
```
pip install channels
```

- **Uninstall using pip -**

```
pip uninstall channels
```

Adding Channels to Django Project -

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



ARFAT
Date 10-1-22
Page 6

Configuring asgi.py -

```
from channels.routing import ProtocolTypeRouter
application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    # Just HTTP for now. (We can add other protocols later.)
})
```

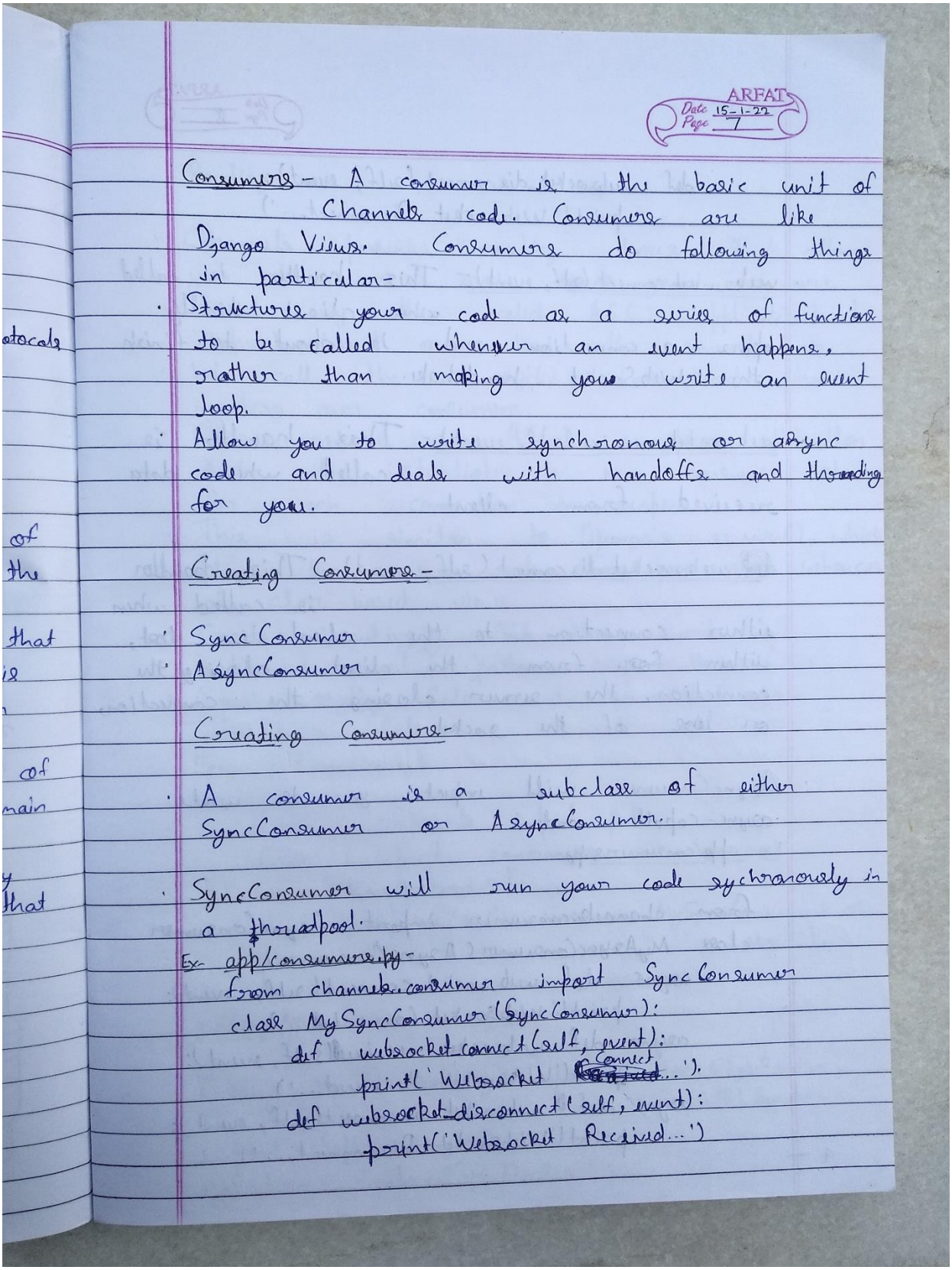
ProtocolTypeRouter -

ProtocolTypeRouter lets you dispatch to one of a number of other ASGI applications on the type value present in the scope.
Protocols will define a fixed type value that their scope contains, so you can use this to distinguish between incoming connection types.
ProtocolTypeRouter should be the top level of your ASGI application stack and the main entry in your routing file.
It takes a single argument - a dictionary mapping type names to ASGI applications that serve them -

```
ProtocolTypeRouter({
    "http": some_app,
    "websocket": some_other_app,
})
```

Configuring settings.py -

```
ASGI_APPLICATION = "myproject.asgi.application"
```



ARFAT
Date 15-1-22
Page 7

Consumers - A consumer is the basic unit of Channels code. Consumers are like Django Views. Consumers do following things in particular -

- Structures your code as a series of functions to be called whenever an event happens, rather than making you write an event loop.
- Allow you to write synchronous or async code and deals with handoffs and threading for you.

Creating Consumers -

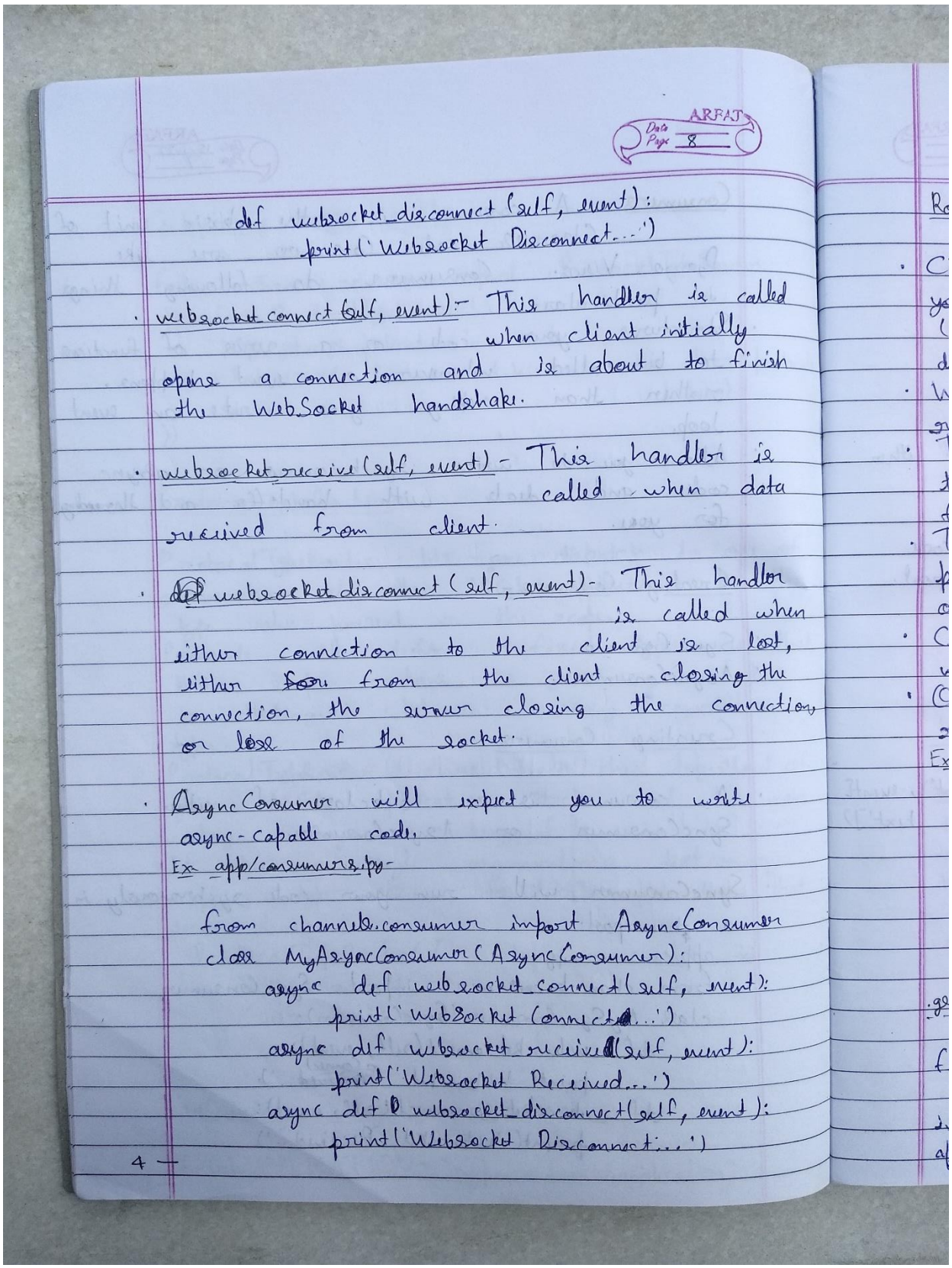
- SyncConsumer
- AsyncConsumer

Creating Consumers -

- A consumer is a subclass of either SyncConsumer or AsyncConsumer.
- SyncConsumer will run your code synchronously in a threadpool.

Ex. app/consumers.py -

```
from channels.consumer import SyncConsumer
class MySyncConsumer(SyncConsumer):
    def websocket_connect(self, event):
        print('Websocket Connect...')
    def websocket_disconnect(self, event):
        print('Websocket Received...')
```



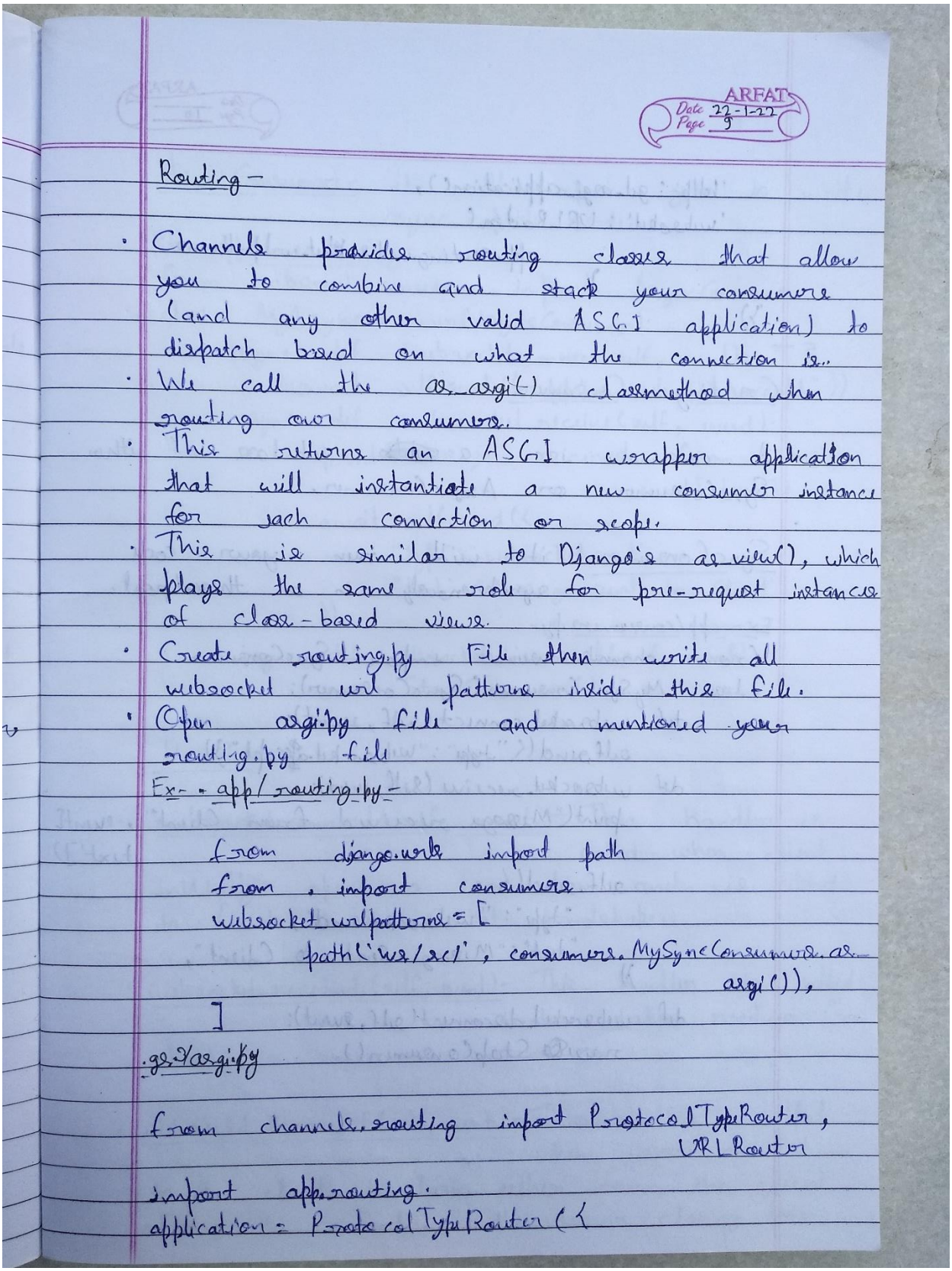
```
def websocket_disconnect(self, event):
    print('Websocket Disconnect...')
```

- websocket_connect(self, event):- This handler is called when client initially opens a connection and is about to finish the WebSocket handshake.
- websocket_receive(self, event) - This handler is called when data received from client.
- websocket_disconnect(self, event)- This handler is called when either connection to the client is lost, either from the client closing the connection, the server closing the connection or loss of the socket.
- AsyncConsumer will expect you to write async-capable code.

Ex app/consumers.py-

```
from channels.consumer import AsyncConsumer
class MyAsyncConsumer(AsyncConsumer):
    async def websocket_connect(self, event):
        print('WebSocket Connected...')
    async def websocket_received(self, event):
        print('Websocket Received...')
    async def websocket_disconnect(self, event):
        print('Websocket Disconnect...')
```

4 —



ARFAT
Date 22-1-22
Page 9

Routing -

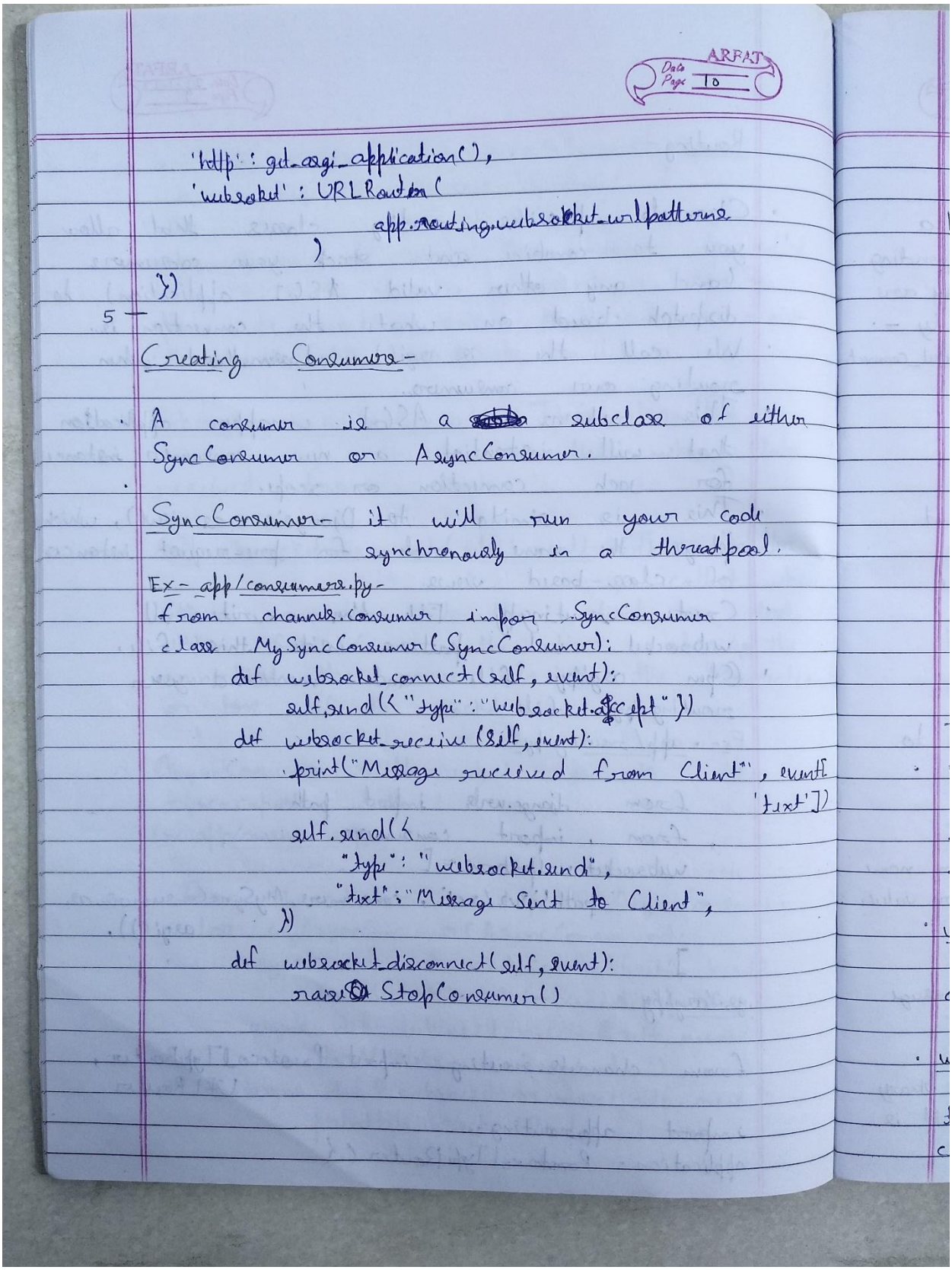
- Channels provides routing classes that allow you to combine and stack your consumers (and any other valid ASGI application) to dispatch based on what the connection is..
- We call the as_asgi() classmethod when routing our consumers.
- This returns an ASGI wrapper application that will instantiate a new consumer instance for each connection or scope.
- This is similar to Django's as_view(), which plays the same role for per-request instances of class-based views.
- Create routing.py File then write all websocket url patterns inside this file.
- Open asgi.py file and mentioned your routing.py file

Ex- - app/routing.py -

```
from django.urls import path
from . import consumers
websocket_urlpatterns = [
    path('ws/sc/', consumers.MySyncConsumer.as_
                    asgi()),
]
```

.gs/asgi.py

```
from channels.routing import ProtocolTypeRouter,
                             URLRouter
import app.routing.
application = ProtocolTypeRouter({
```



```
    'http': get_asgi_application(),
    'websocket': URLRouter(
            app.routing.websocket_urlpatterns
        )
})
```

5 —

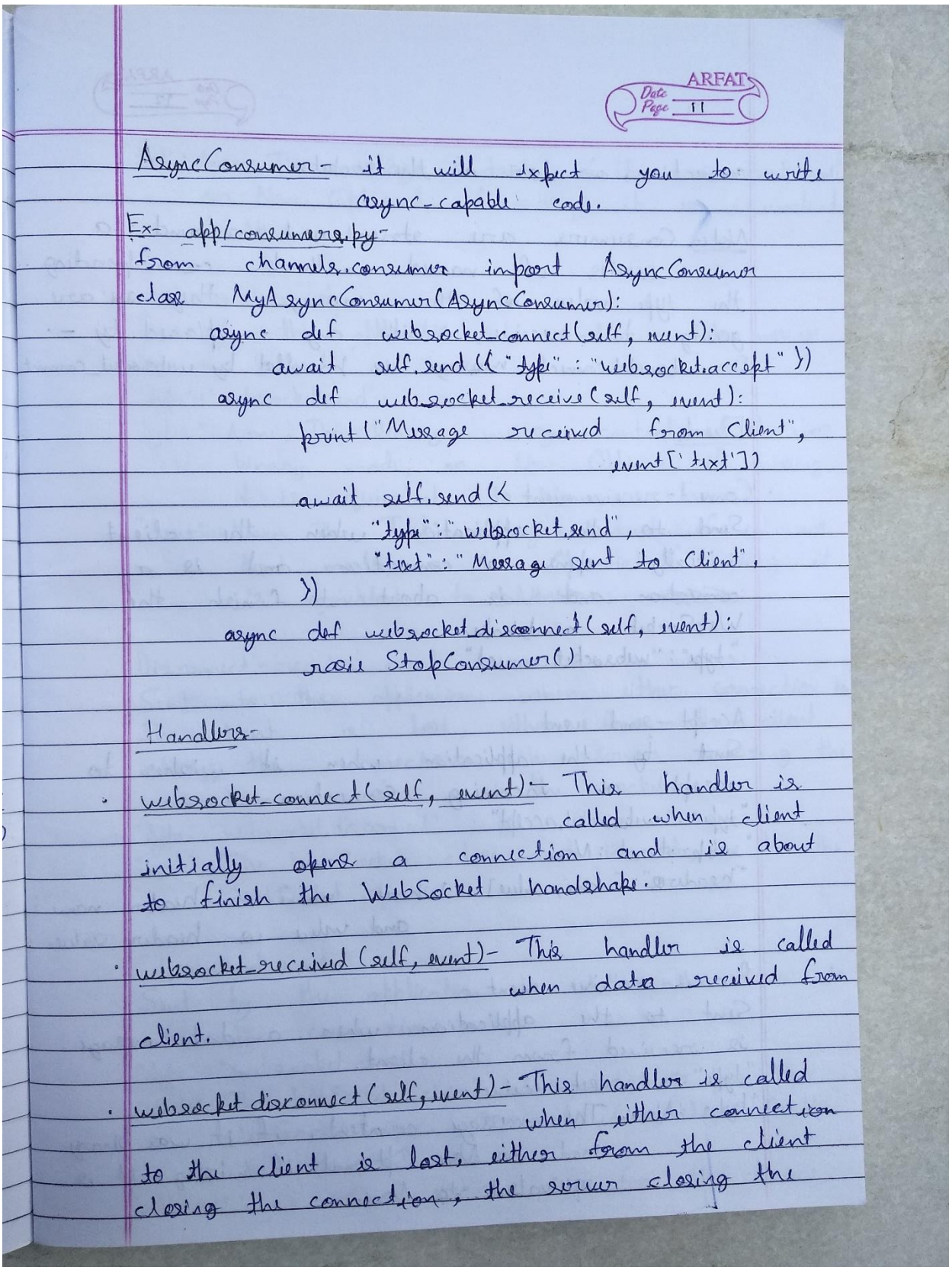
Creating Consumers -

A consumer is a subclass of either SyncConsumer or AsyncConsumer.

SyncConsumer - it will run your code synchronously in a threadpool.

Ex - app/consumers.py -

```
from channels.consumer import SyncConsumer
class MySyncConsumer(SyncConsumer):
    def websocket_connect(self, event):
        self.send({"type": "websocket.accept"})
    def websocket_receive(self, event):
        print("Message received from Client", event['text'])
        self.send({
            "type": "websocket.send",
            "text": "Message Sent to Client",
        })
    def websocket_disconnect(self, event):
        raise StopConsumer()
```



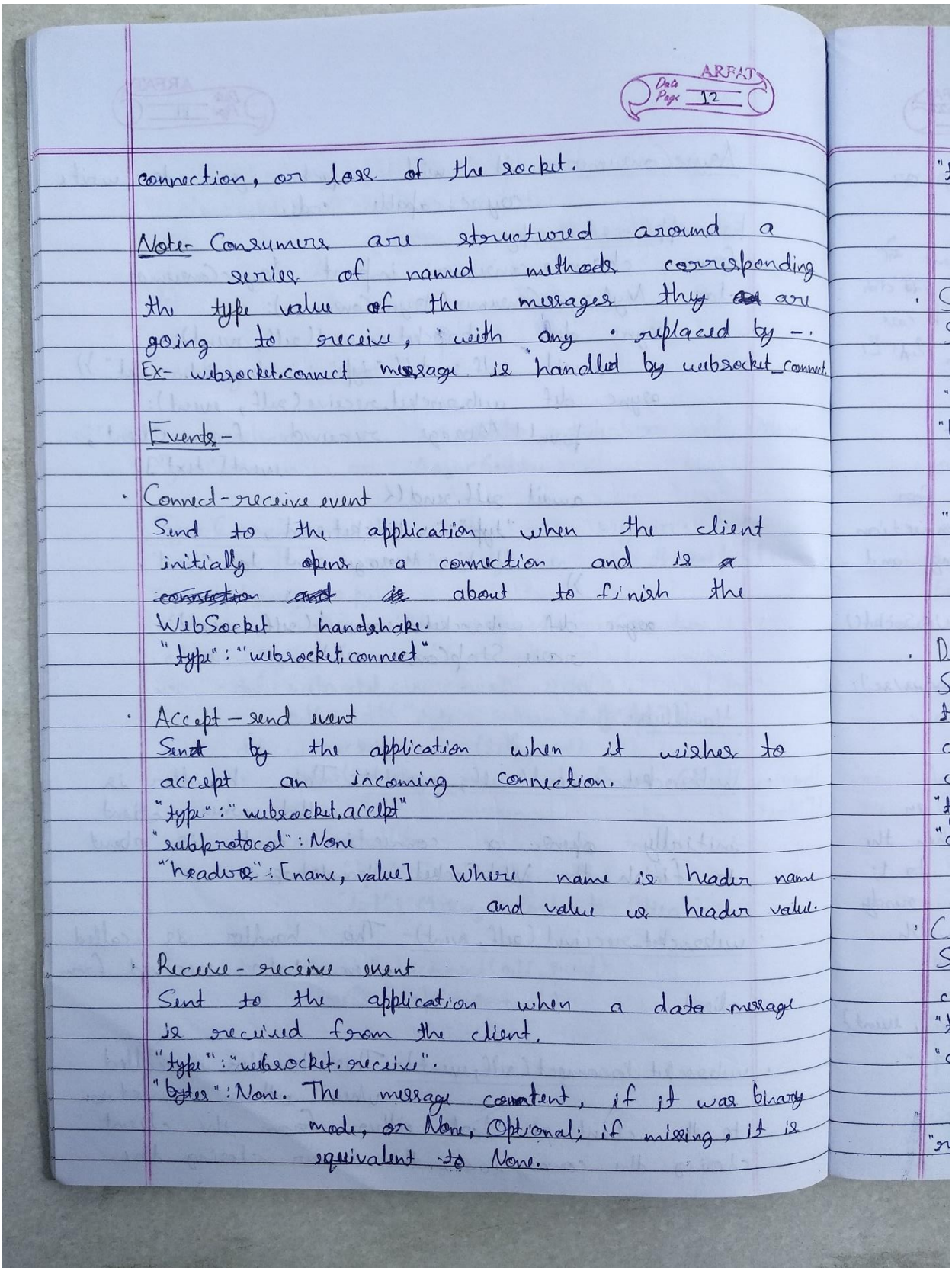
AsyncConsumer - it will expect you to write async-capable code.

Ex- app/consumers.py

```
from channels.consumer import AsyncConsumer
class MyAsyncConsumer(AsyncConsumer):
    async def websocket_connect(self, event):
        await self.send({"type": "websocket.accept"})
    async def websocket_receive(self, event):
        print("Message received from Client",
                                    event['text'])
        await self.send({
            "type": "websocket.send",
            "text": "Message sent to Client",
        })
    async def websocket_disconnect(self, event):
        raise StopConsumer()
```

Handlers-

- websocket_connect(self, event) - This handler is called when client initially opens a connection and is about to finish the WebSocket handshake.
- websocket_received(self, event) - This handler is called when data received from client.
- websocket_disconnect(self, event) - This handler is called when either connection to the client is lost, either from the client closing the connection, the server closing the



connection, or loss of the socket.

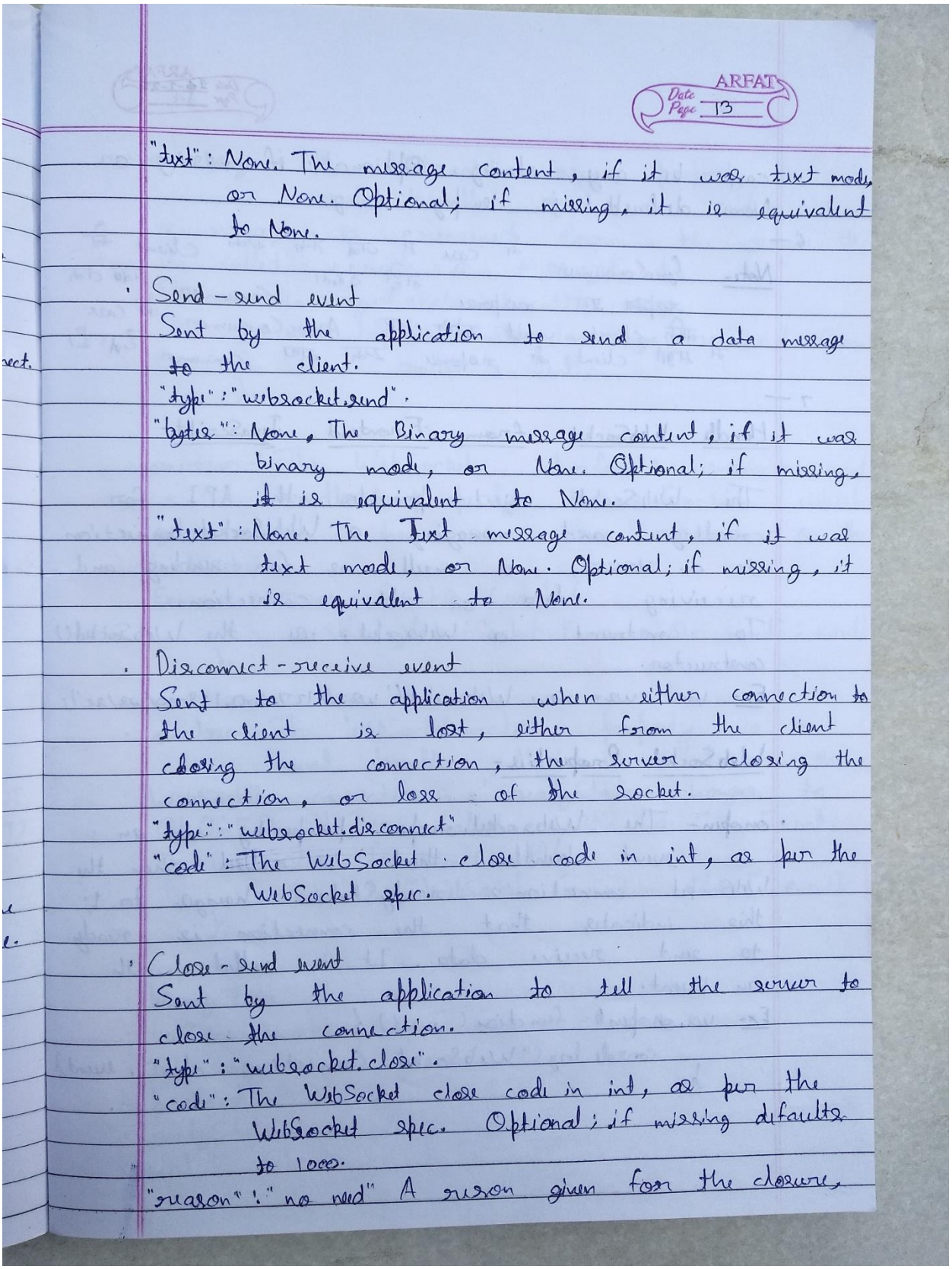
Note- Consumers are structured around a series of named methods corresponding the type value of the messages they are going to receive, with any . replaced by _.
Ex- websocket.connect message is handled by websocket_connect.

Events -

- Connect - receive event
  Send to the application when the client initially opens a connection and is about to finish the WebSocket handshake.
  "type": "websocket.connect"

- Accept - send event
  Sent by the application when it wishes to accept an incoming connection.
  "type": "websocket.accept"
  "subprotocol": None
  "headers": [name, value] Where name is header name and value is header value.

- Receive - receive event
  Sent to the application when a data message is received from the client.
  "type": "websocket.receive".
  "bytes": None. The message content, if it was binary mode, or None. Optional; if missing, it is equivalent to None.



"text": None. The message content, if it was text mode, or None. Optional; if missing, it is equivalent to None.

- Send - send event

Sent by the application to send a data message to the client.

"type": "websocket.send".

"bytes": None, The Binary message content, if it was binary mode, or None. Optional; if missing, it is equivalent to None.

"text": None. The Text message content, if it was text mode, or None. Optional; if missing, it is equivalent to None.

- Disconnect - receive event

Sent to the application when either connection to the client is lost, either from the client closing the connection, the server closing the connection, or loss of the socket.

"type": "websocket.disconnect"

"code": The WebSocket close code in int, as per the WebSocket spec.

- Close - send event

Sent by the application to tell the server to close the connection.

"type": "websocket.close".

"code": The WebSocket close code in int, as per the WebSocket spec. Optional; if missing defaults to 1000.

"reason": "no reason" A reason given for the closure.

can be any string. Optional: if missing or None default is empty string.

6 —

Note- SyncConsumer के case में जब तक एक client को response नहीं देदेता तबतक Consumer वाकि client wait करेंगे पर AsyncConsumer के case में सभी client को response एक साथ consumer देता है

7 —

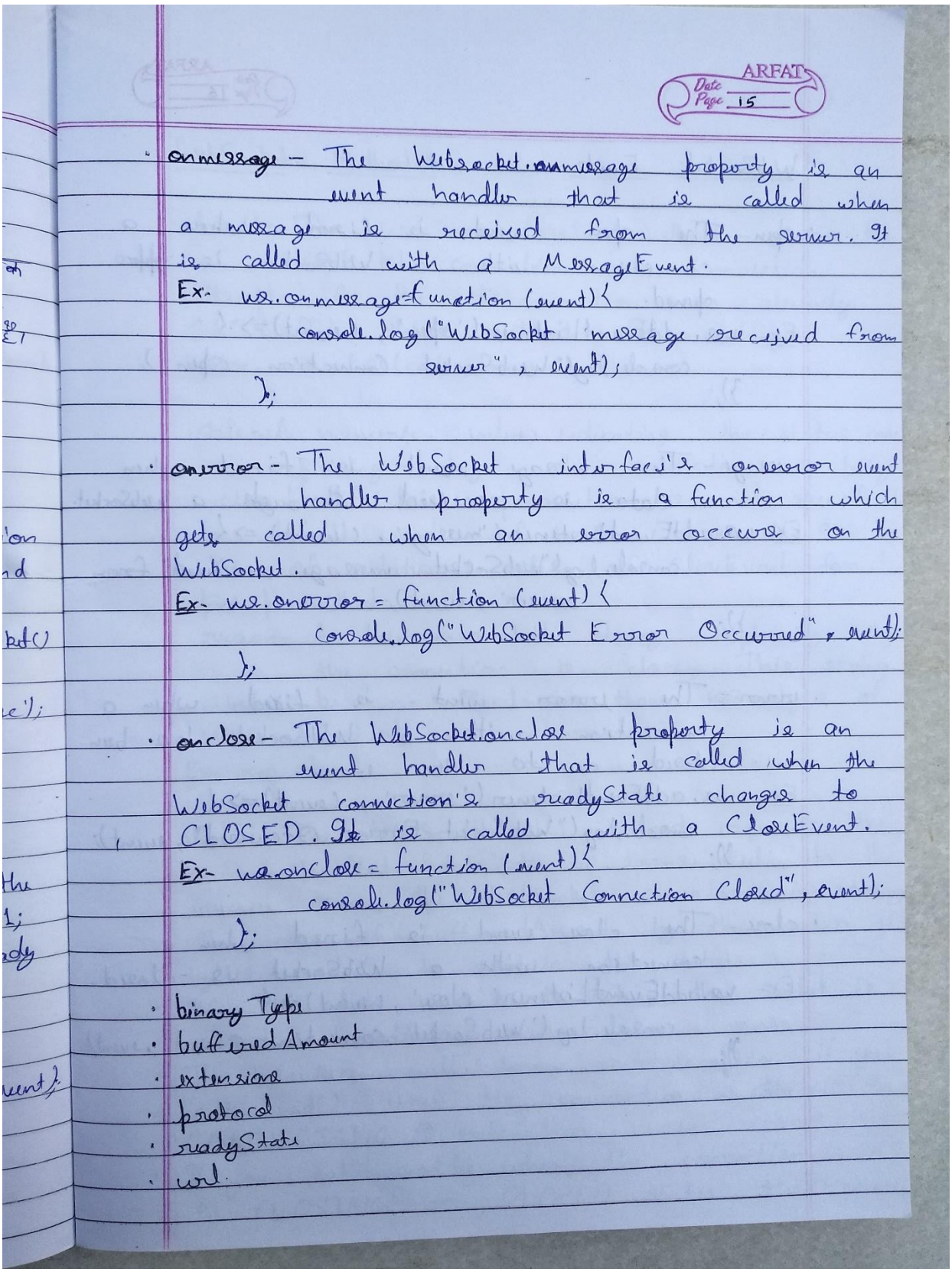
Handle WebSocket from Frontend JavaScript-

The WebSocket object provides the API for creating and managing a WebSocket connection to a server, as well as for sending and receiving data on the connection.
To construct a websocket, use the WebSocket() constructor.
Ex- var ws = new Websocket('ws://127.0.0.1:8000/ws/sc/');

WebSocket Properties-

- onopen- The Websocket.onopen property is an event handler that is called when the WebSocket connection's readyState changes to 1; this indicates that the connection is ready to send receive data. It is called with an Event.

```
Ex- ws.onopen = function (event) {
        console.log("WebSocket Connection open", event)
    };
```



- onmessage – The WebSocket.onmessage property is an event handler that is called when a message is received from the server. It is called with a MessageEvent.

Ex. ws.onmessage=function (event){
console.log("WebSocket message received from server", event);
};

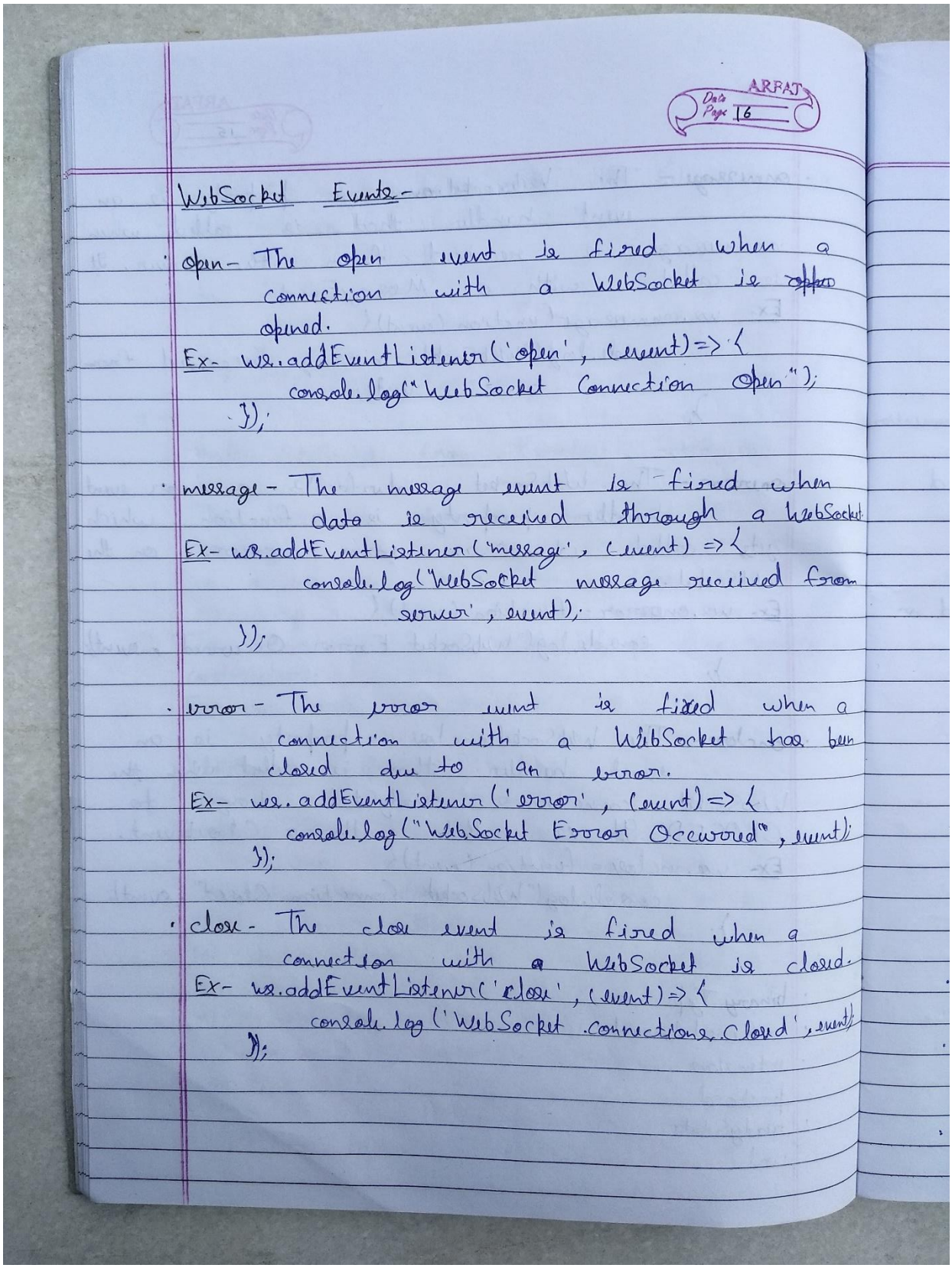
- onerror – The WebSocket interface's onerror event handler property is a function which gets called when an error occurs on the WebSocket.

Ex. ws.onerror = function (event) {
console.log("WebSocket Error Occurred", event);
};

- onclose – The WebSocket.onclose property is an event handler that is called when the WebSocket connection's readyState changes to CLOSED. It is called with a CloseEvent.

Ex- ws.onclose = function (event){
console.log("WebSocket Connection Closed", event);
};

- binaryType
- bufferedAmount
- extensions
- protocol
- readyState
- url



WebSocket Events -

- open - The open event is fired when a connection with a WebSocket is opened.

Ex- ws.addEventListener('open', (event) => {
    console.log("WebSocket Connection Open");
});

- message - The message event is fired when data is received through a WebSocket.

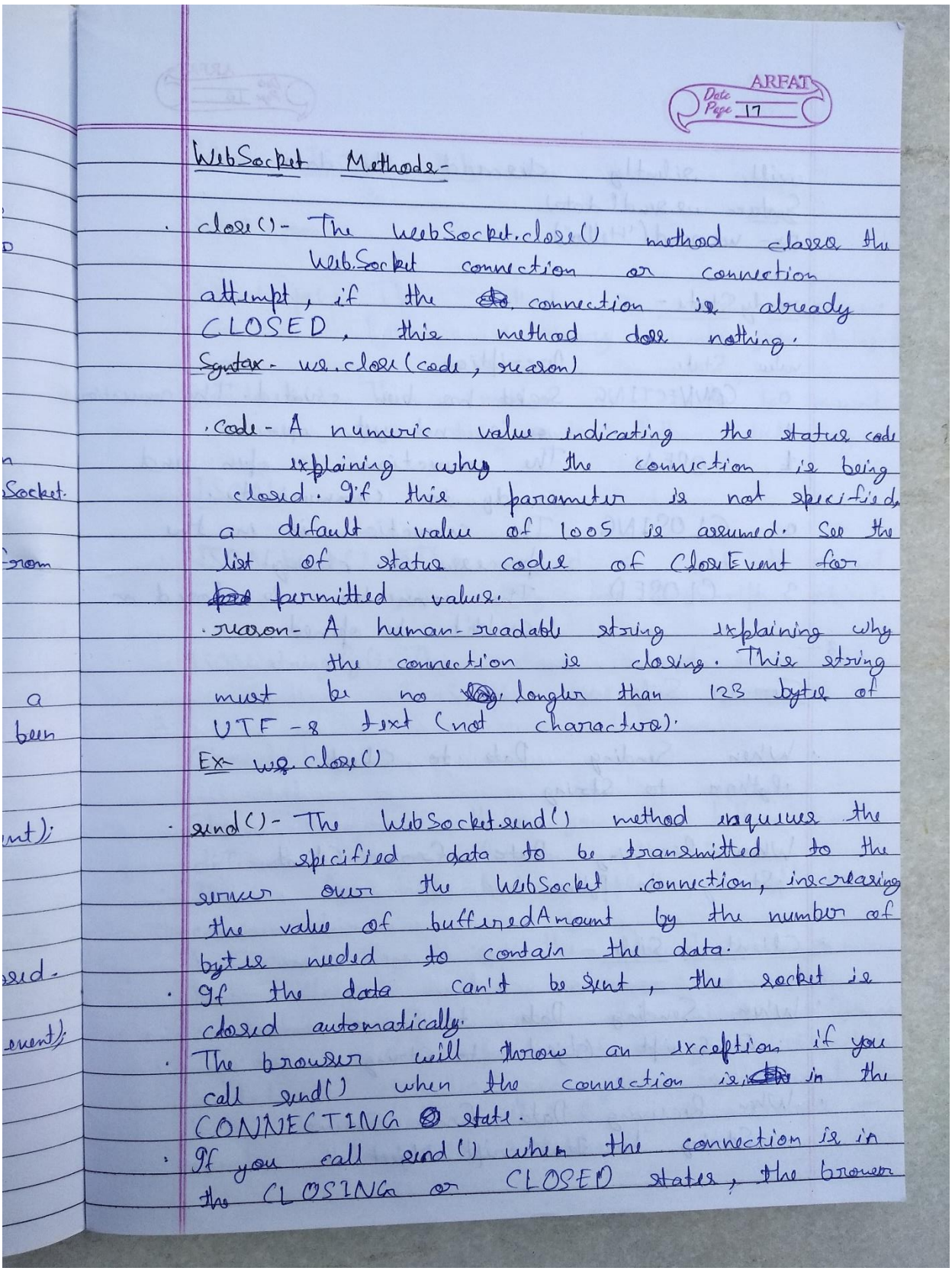
Ex- ws.addEventListener('message', (event) => {
    console.log('WebSocket message received from server', event);
});

- error - The error event is fired when a connection with a WebSocket has been closed due to an error.

Ex- ws.addEventListener('error', (event) => {
    console.log("WebSocket Error Occurred", event);
});

- close - The close event is fired when a connection with a WebSocket is closed.

Ex- ws.addEventListener('close', (event) => {
    console.log('WebSocket Connection Closed', event);
});



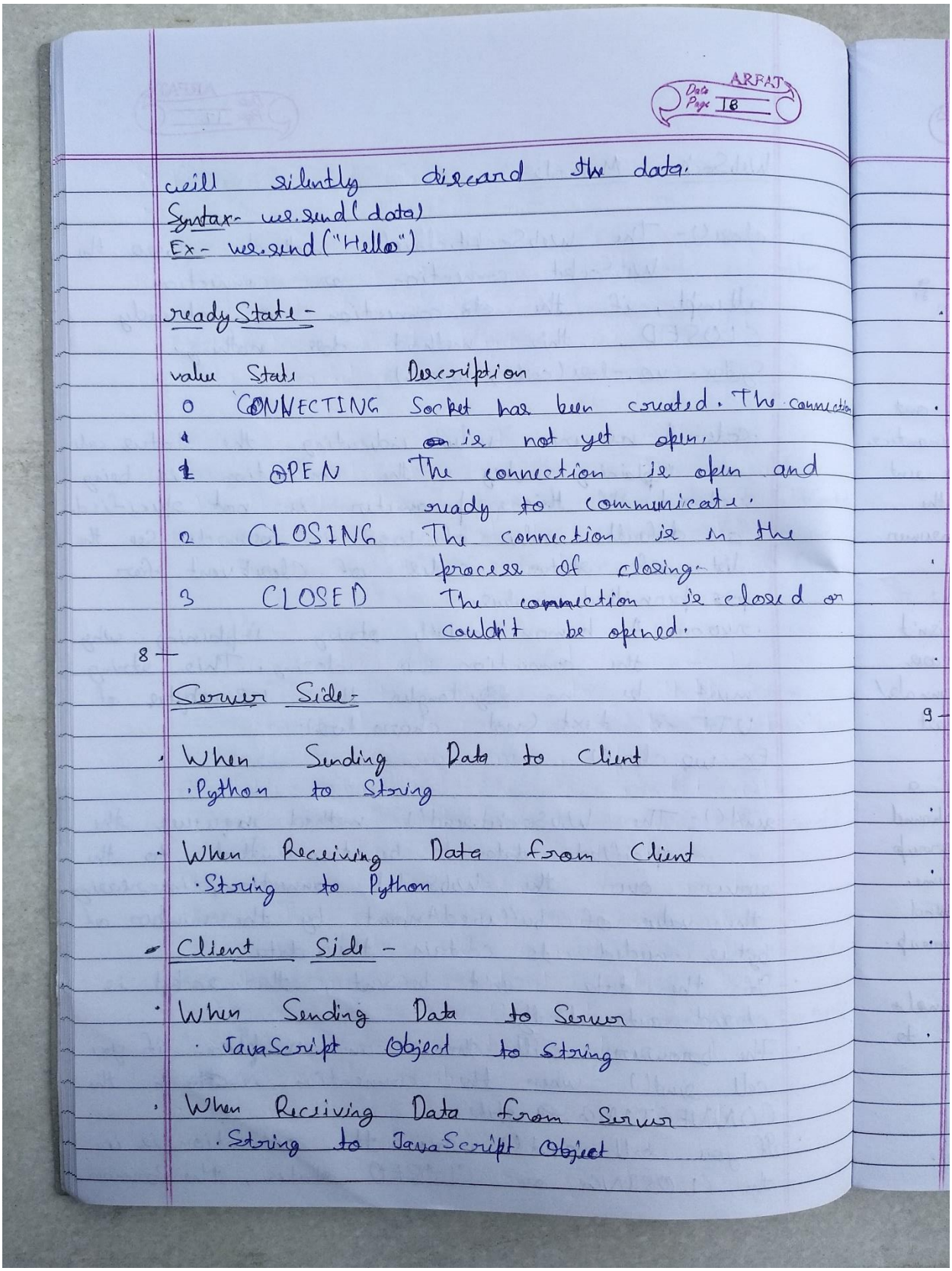
ARFAT
Date Page 17

WebSocket Methods-

- close()- The WebSocket.close() method closes the WebSocket connection or connection attempt, if the connection is already CLOSED, this method does nothing.
Syntax- ws.close(code, reason)

- code - A numeric value indicating the status code explaining why the connection is being closed. If this parameter is not specified, a default value of 1005 is assumed. See the list of status codes of CloseEvent for permitted values.
- reason - A human-readable string explaining why the connection is closing. This string must be no longer than 123 bytes of UTF-8 text (not characters).
Ex- ws.close()

- send()- The WebSocket.send() method enqueues the specified data to be transmitted to the server over the WebSocket connection, increasing the value of bufferedAmount by the number of bytes needed to contain the data.
- If the data can't be sent, the socket is closed automatically.
- The browser will throw an exception if you call send() when the connection is in the CONNECTING state.
- If you call send() when the connection is in the CLOSING or CLOSED states, the browser



will silently discard the data.
Syntax- ws.send(data)
Ex- ws.send("Hello")

readyState -

value	State	Description
0	CONNECTING	Socket has been created. The connection is not yet open.
1	OPEN	The connection is open and ready to communicate.
2	CLOSING	The connection is in the process of closing.
3	CLOSED	The connection is closed or couldn't be opened.

8 —

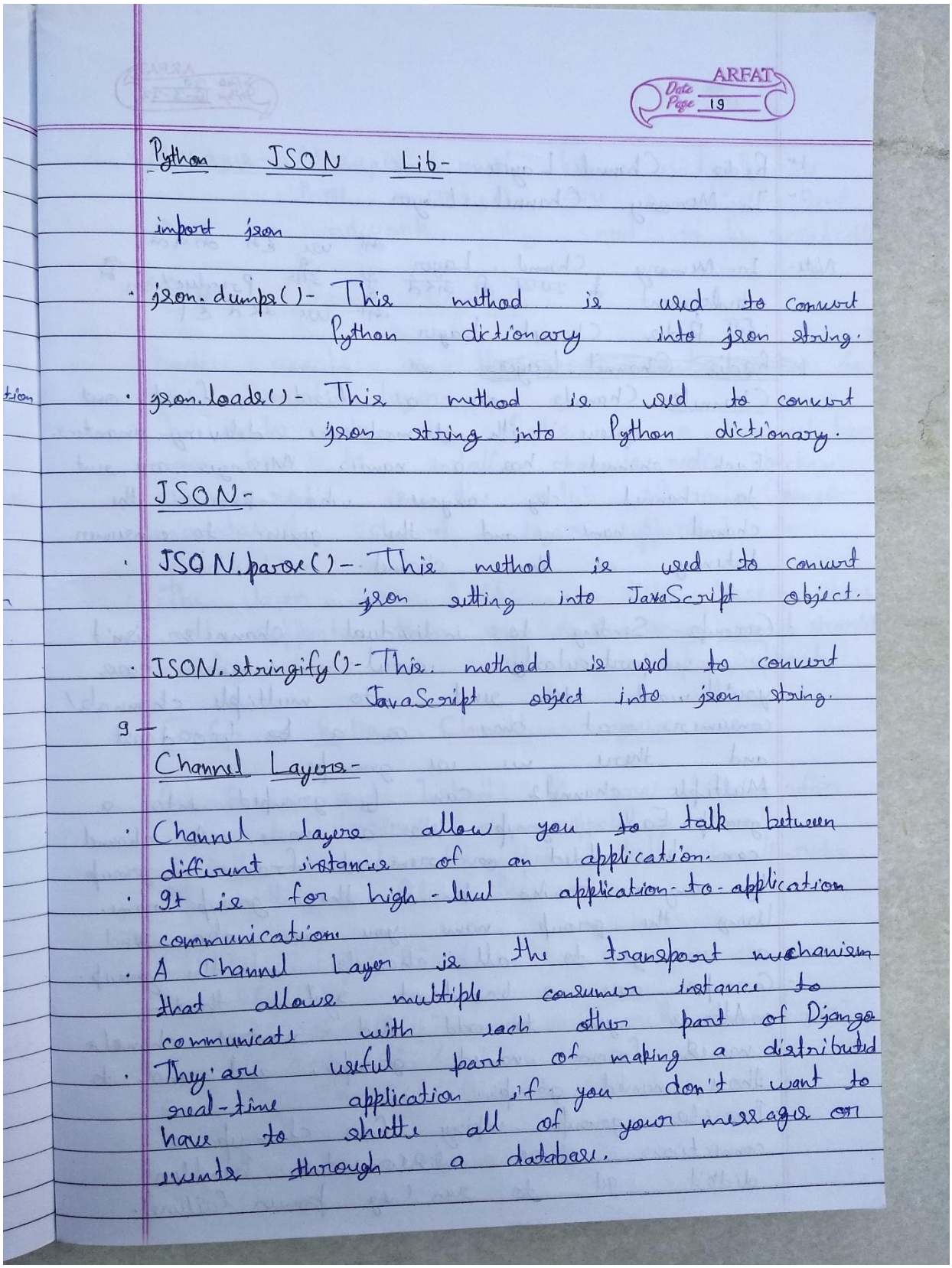
Server Side -

9 —

- When Sending Data to Client
  - Python to String
- When Receiving Data from Client
  - String to Python

Client Side -

- When Sending Data to Server
  - JavaScript Object to String
- When Receiving Data from Server
  - String to JavaScript Object



ARFAT Date Page 19

Python JSON Lib-

import json

- json.dumps()- This method is used to convert Python dictionary into json string.
- json.loads()- This method is used to convert json string into Python dictionary.

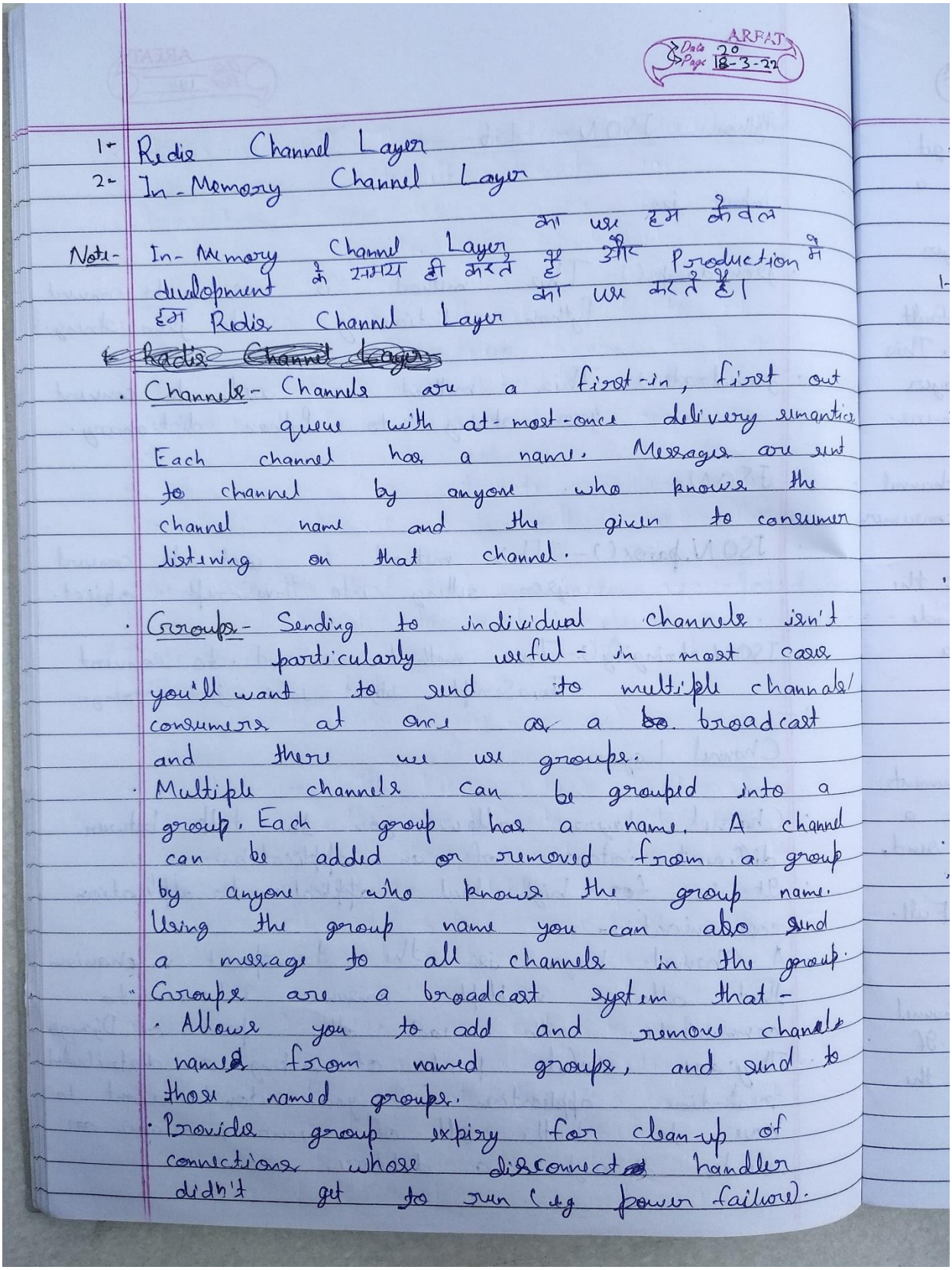
JSON-

- JSON.parse()- This method is used to convert json string into JavaScript object.
- JSON.stringify()- This method is used to convert JavaScript object into json string.

9-

Channel Layers-

- Channel layers allow you to talk between different instances of an application.
- It is for high-level application-to-application communication.
- A Channel Layer is the transport mechanism that allows multiple consumer instance to communicate with each other part of Django.
- They are useful part of making a distributed real-time application if you don't want to have to shuttle all of your messages or events through a database.



ARFAT
Date 20
Page 18-3-22

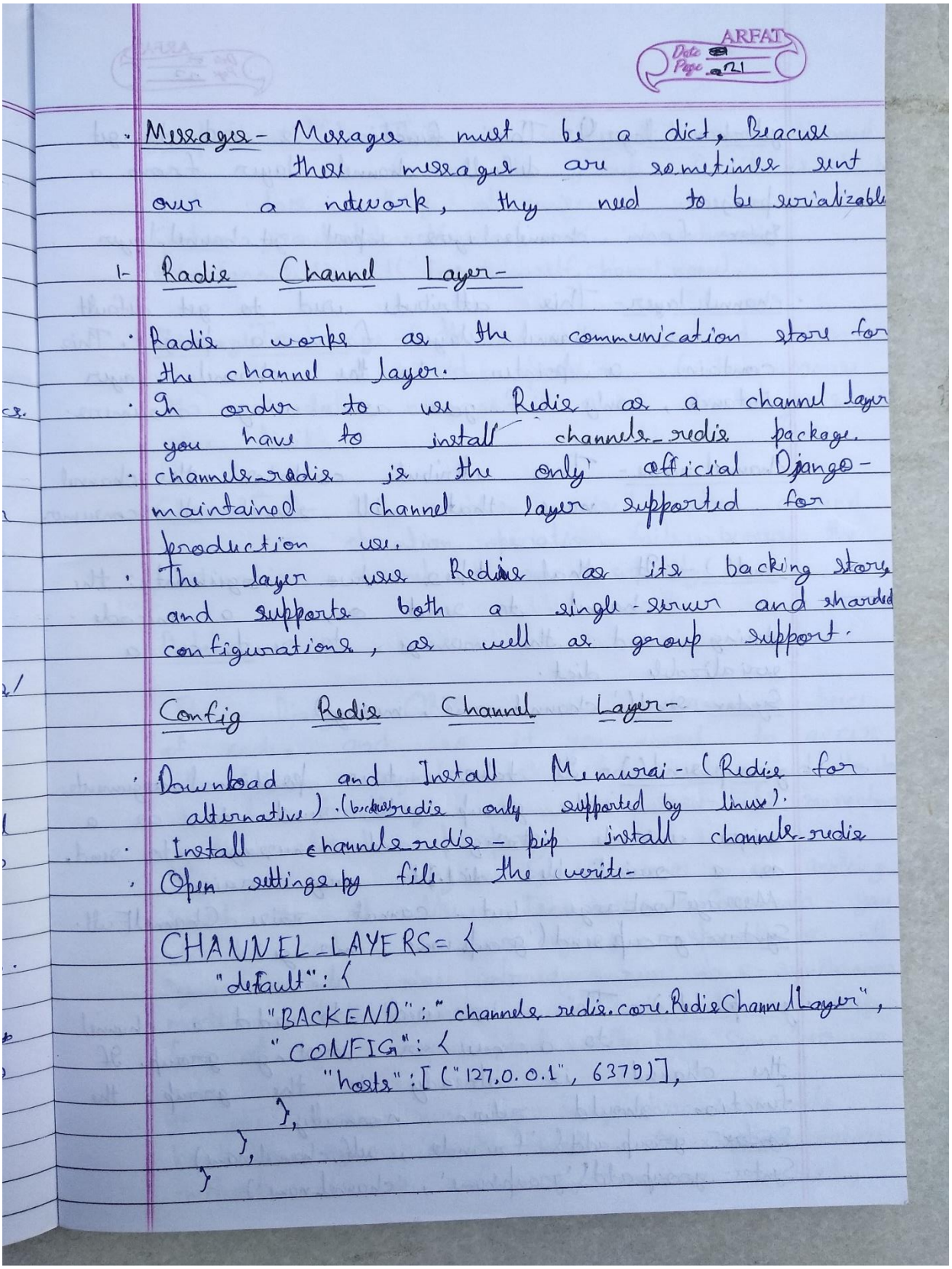
1- Redis Channel Layer
2- In-Memory Channel Layer

Note- In-Memory Channel Layer को use हम करते हैं development के समय ही करते हैं और Production में हम Redis Channel Layer को use करते हैं।

~~Redis Channel Layer~~

- Channels- Channels are a first-in, first out queue with at-most-once delivery semantics. Each channel has a name. Messages are sent to channel by anyone who knows the channel name and the given to consumer listening on that channel.

- Groups- Sending to individual channels isn't particularly useful - in most case you'll want to send to multiple channels/ consumers at once or a ~~bo~~ broadcast and there we use groups.
- Multiple channels can be grouped into a group. Each group has a name. A channel can be added or removed from a group by anyone who knows the group name. Using the group name you can also send a message to all channels in the group.
- Groups are a broadcast system that -
  - Allows you to add and remove channels named from named groups, and send to those named groups.
  - Provides group expiry for clean-up of connections whose disconnect handler didn't get to run (eg power failure).



- Messages - Messages must be a dict, Because these messages are sometimes sent over a network, they need to be serializable

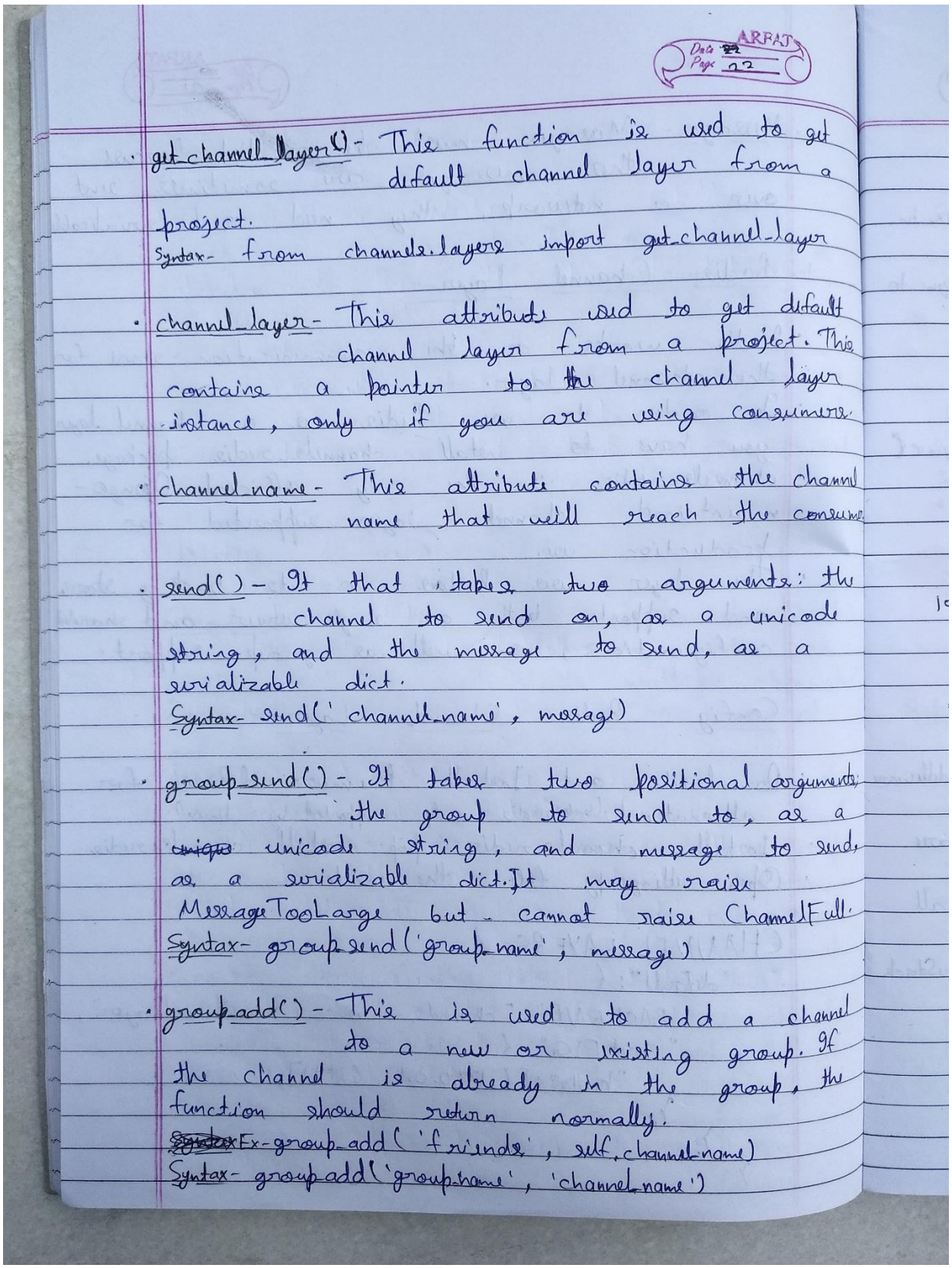
1- Redis Channel Layer -

- Redis works as the communication store for the channel layer.
- In order to use Redis as a channel layer you have to install channels_redis package.
- channels_redis is the only official Django-maintained channel layer supported for production use.
- The layer uses Redis as its backing store, and supports both a single-server and sharded configurations, as well as group support.

Config Redis Channel Layer -

- Download and Install Memurai - (Redis for alternative). (because redis only supported by linux).
- Install channels_redis - pip install channels_redis
- Open settings.py file the write -

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("127.0.0.1", 6379)],
        },
    },
}
```



- get_channel_layer() - This function is used to get default channel layer from a project.
  Syntax- from channels.layers import get_channel_layer

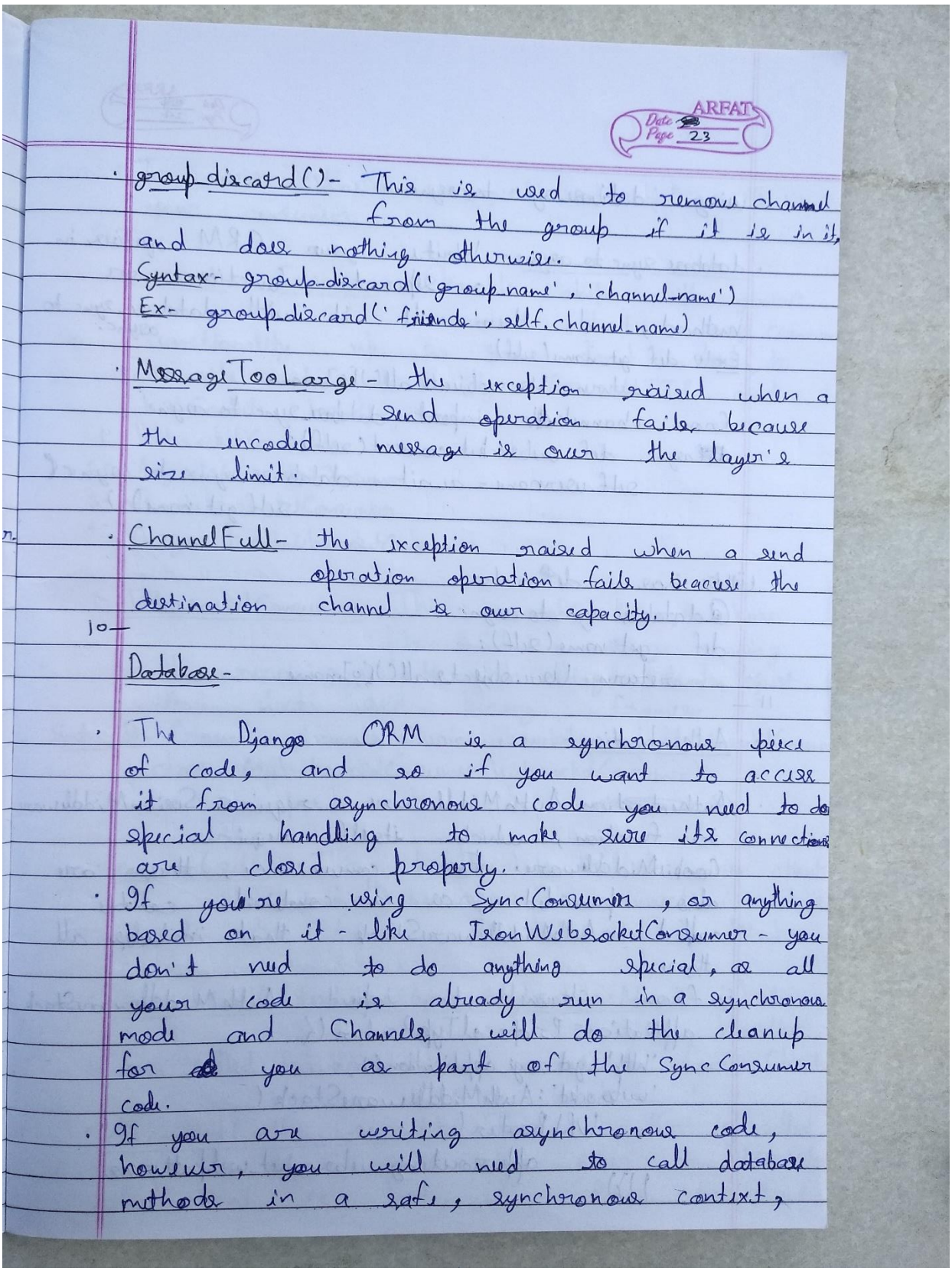
- channel_layer - This attribute used to get default channel layer from a project. This contains a pointer to the channel layer instance, only if you are using consumers.

- channel_name - This attribute contains the channel name that will reach the consumer.

- send() - It that takes two arguments: the channel to send on, as a unicode string, and the message to send, as a serializable dict.
  Syntax- send('channel_name', message)

- group_send() - It takes two positional arguments: the group to send to, as a unicode string, and message to send, as a serializable dict. It may raise MessageTooLarge but cannot raise ChannelFull.
  Syntax- group_send('group_name', message)

- group_add() - This is used to add a channel to a new or existing group. If the channel is already in the group, the function should return normally.
  Ex- group_add('friends', self.channel_name)
  Syntax- group_add('group_name', 'channel_name')



- group_discard() - This is used to remove channel from the group if it is in it, and does nothing otherwise.
  Syntax - group_discard('group_name', 'channel_name')
  Ex - group_discard('friends', self.channel_name)

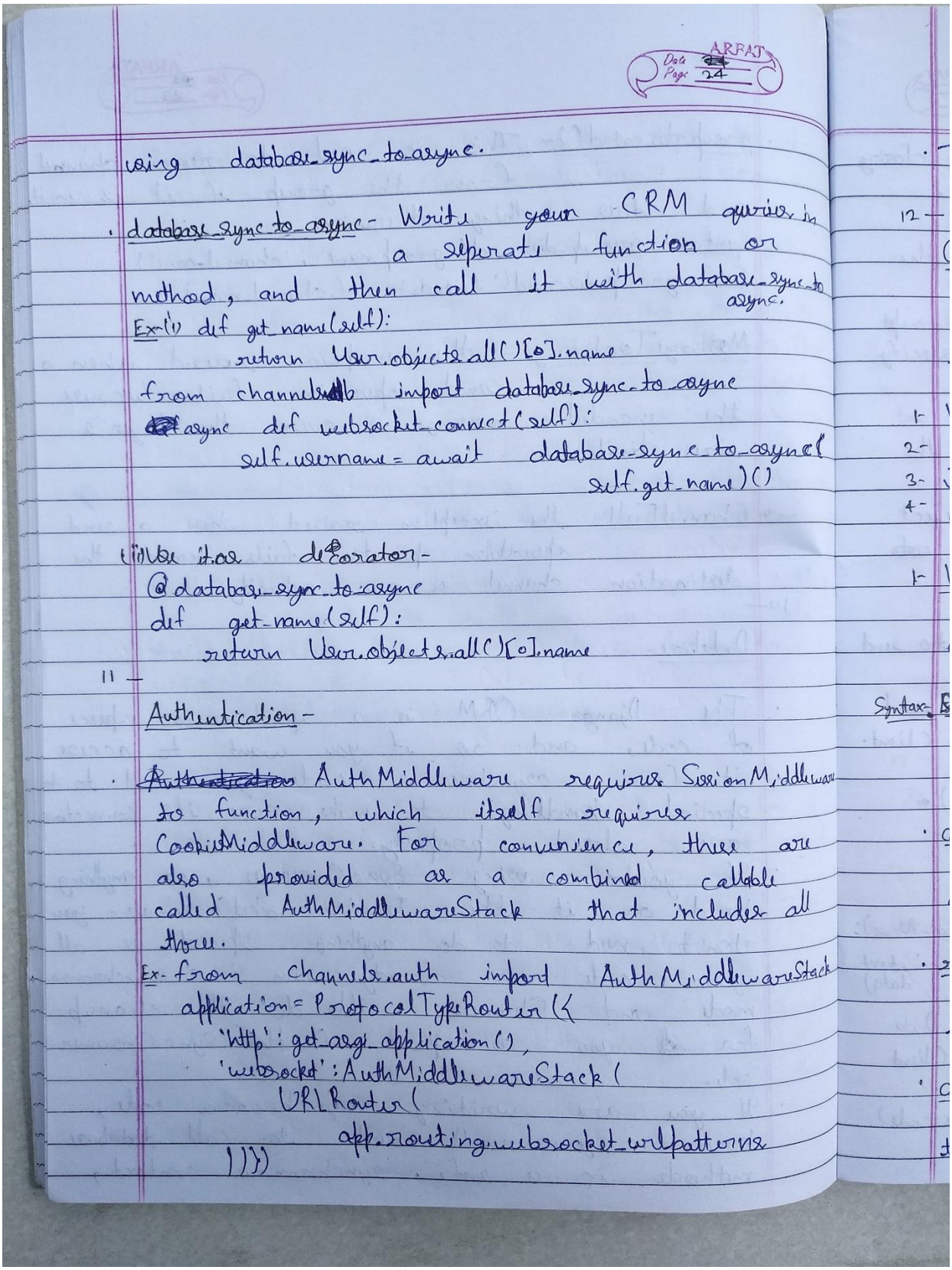
- MessageTooLarge - the exception raised when a send operation fails because the encoded message is over the layer's size limit.

- ChannelFull - the exception raised when a send operation operation fails because the destination channel is over capacity.

10 —

Database -

- The Django ORM is a synchronous piece of code, and so if you want to access it from asynchronous code you need to do special handling to make sure its connections are closed properly.
- If you're using SyncConsumer, or anything based on it - like JsonWebsocketConsumer - you don't need to do anything special, as all your code is already run in a synchronous mode and Channels will do the cleanup for you as part of the SyncConsumer code.
- If you are writing asynchronous code, however, you will need to call database methods in a safe, synchronous context,



using database_sync_to_async.

. database_sync_to_async - Write your CRM queries in a separate function or method, and then call it with database_sync_to_async.

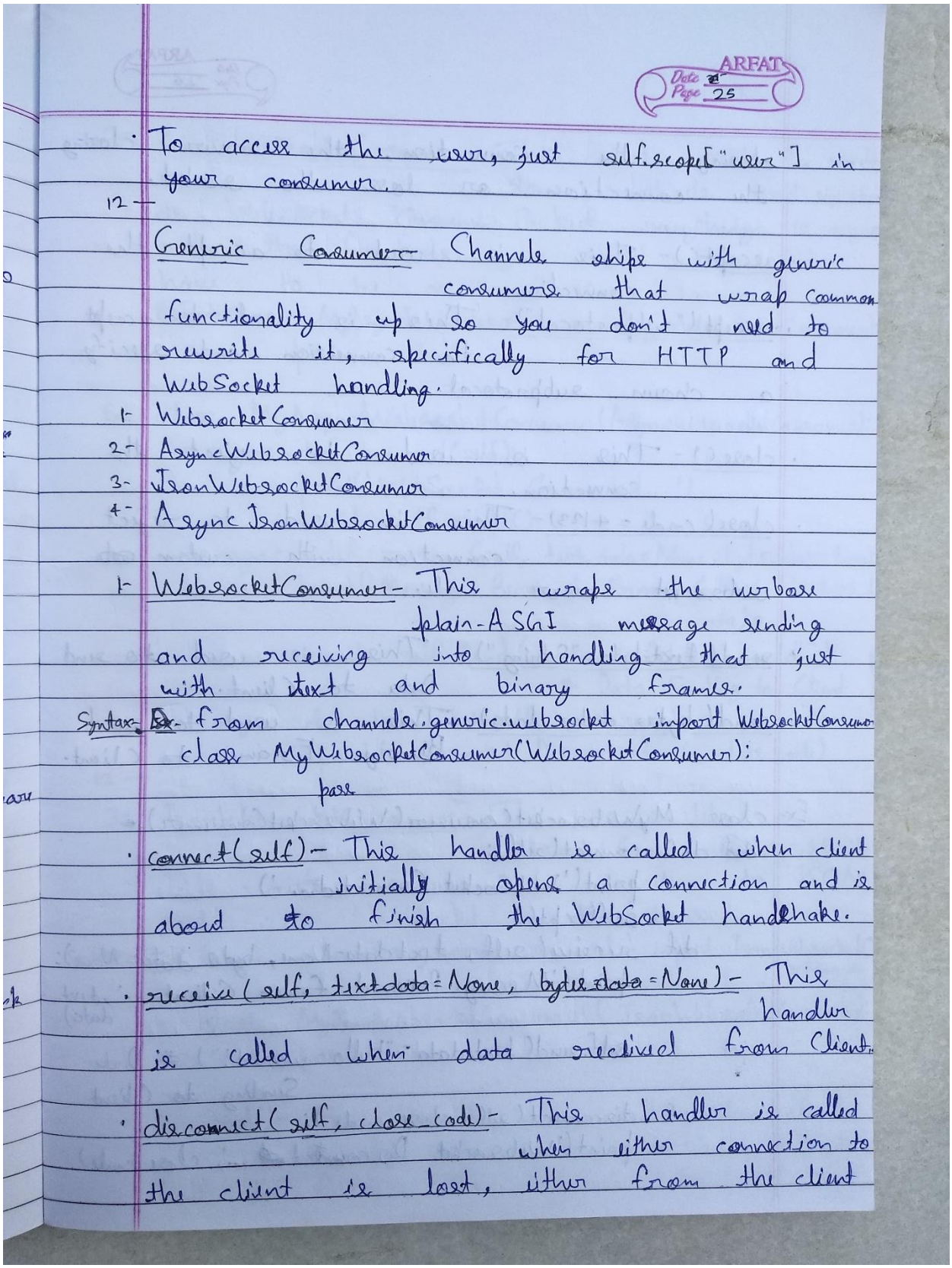
Ex-(i) def get_name(self):
    return User.objects.all()[0].name
from channels.db import database_sync_to_async
async def websocket_connect(self):
    self.username = await database_sync_to_async(
        self.get_name)()

(ii) Use it as decorator -
@database_sync_to_async
def get_name(self):
    return User.objects.all()[0].name

Authentication -

. ~~Authentication~~ AuthMiddleware requires SessionMiddleware to function, which itself requires CookieMiddleware. For convenience, these are also provided as a combined callable called AuthMiddlewareStack that includes all three.

Ex- from channels.auth import AuthMiddlewareStack
application = ProtocolTypeRouter({
    'http': get_asgi_application(),
    'websocket': AuthMiddlewareStack(
        URLRouter(
            app.routing.websocket_urlpatterns
    ))})



- To access the user, just self.scope["user"] in your consumer.

12 —

Generic Consumer — Channels ships with generic consumers that wrap common functionality up so you don't need to rewrite it, specifically for HTTP and WebSocket handling.

1- WebsocketConsumer
2- AsyncWebsocketConsumer
3- JsonWebsocketConsumer
4- AsyncJsonWebsocketConsumer

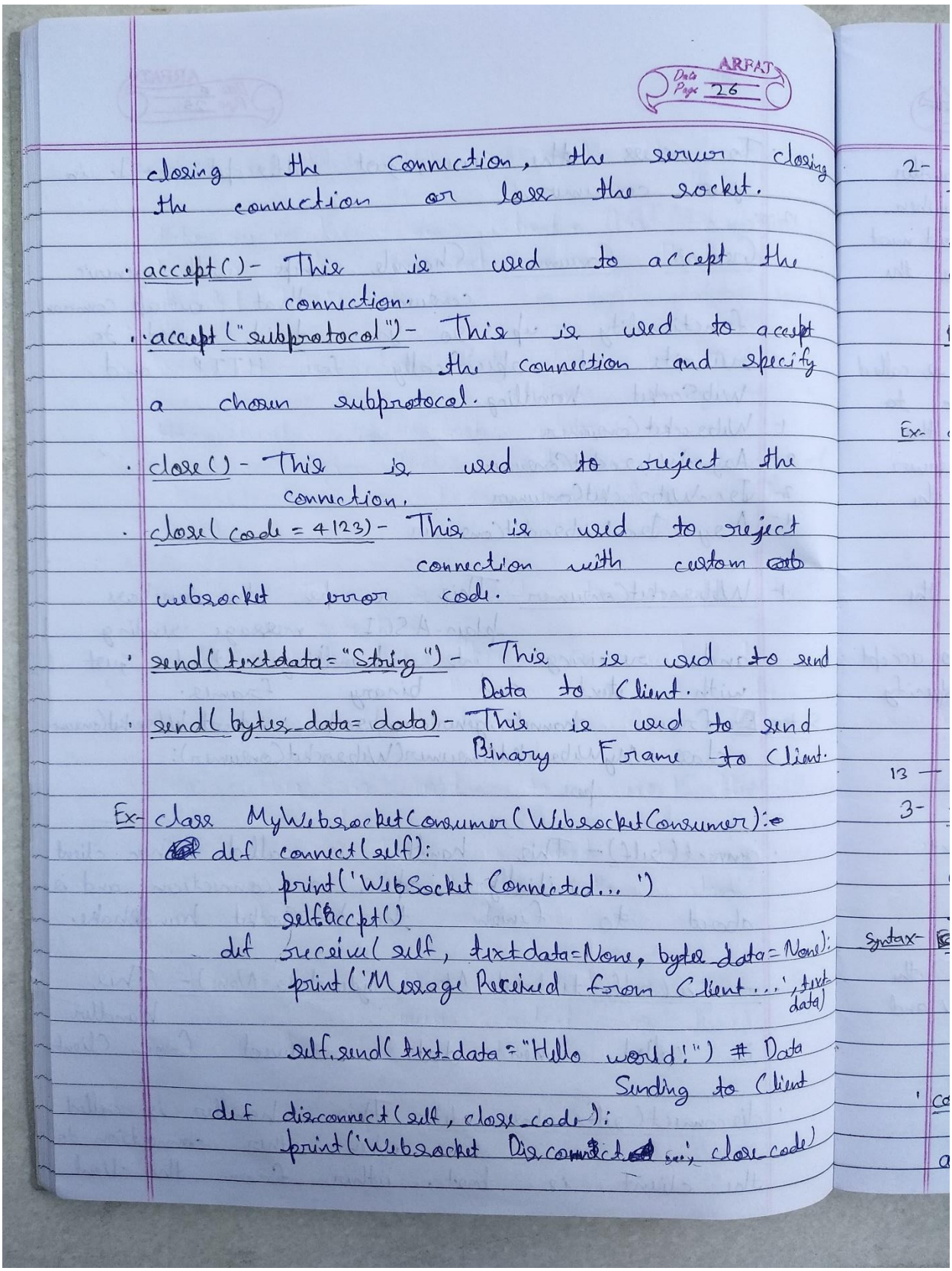
1- WebsocketConsumer - This wraps the verbose plain-ASGI message sending and receiving into handling that just with text and binary frames.

Syntax - from channels.generic.websocket import WebsocketConsumer
class MyWebsocketConsumer(WebsocketConsumer):
    pass

- connect(self) - This handler is called when client initially opens a connection and is about to finish the WebSocket handshake.

- receive(self, text_data=None, bytes_data=None) - This handler is called when data received from client.

- disconnect(self, close_code) - This handler is called when either connection to the client is lost, either from the client

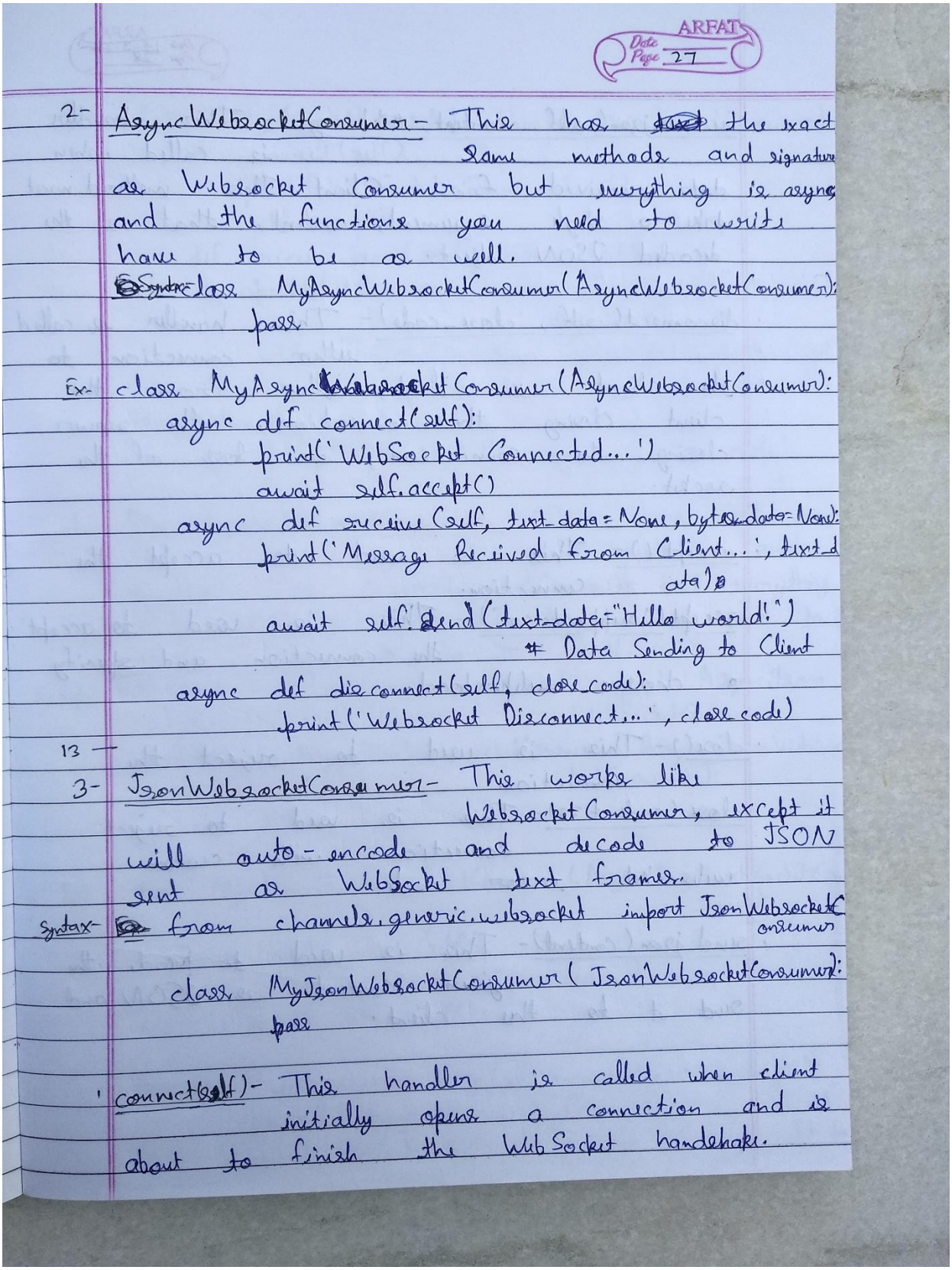


closing the connection, the server closing the connection or lose the socket.

- accept() - This is used to accept the connection.
- accept("subprotocol") - This is used to accept the connection and specify a chosen subprotocol.
- close() - This is used to reject the connection.
- close(code=4123) - This is used to reject connection with custom websocket error code.
- send(text_data="String") - This is used to send Data to Client.
- send(bytes_data=data) - This is used to send Binary Frame to Client.

Ex-

```
class MyWebsocketConsumer(WebsocketConsumer):
    def connect(self):
        print('WebSocket Connected...')
        self.accept()
    def receive(self, text_data=None, bytes_data=None):
        print('Message Received from Client...', text_data)
        self.send(text_data="Hello world!") # Data Sending to Client
    def disconnect(self, close_code):
        print('Websocket Disconnected...', close_code)
```



2- AsyncWebsocketConsumer - This has the exact same methods and signature as WebsocketConsumer but everything is async and the functions you need to write have to be as well.

Syntax-
```
class MyAsyncWebsocketConsumer(AsyncWebsocketConsumer):
    pass
```

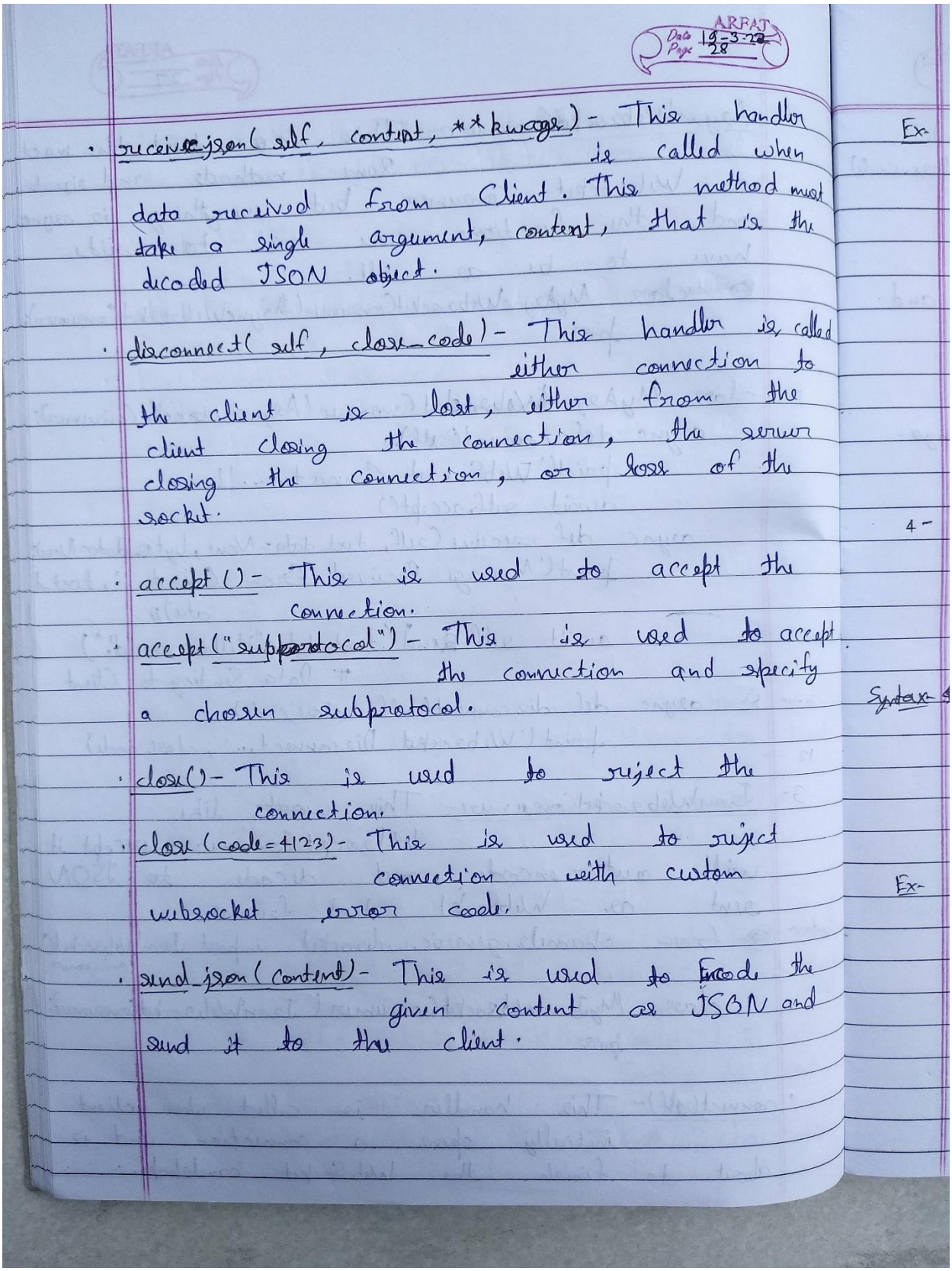
Ex-
```
class MyAsyncWebsocketConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        print('WebSocket Connected...')
        await self.accept()
    async def receive(self, text_data=None, bytes_data=None):
        print('Message Received from Client...', text_data)
        await self.send(text_data="Hello world!")
                    * Data Sending to Client
    async def disconnect(self, close_code):
        print('Websocket Disconnect...', close_code)
```

13 —

3- JsonWebsocketConsumer - This works like WebsocketConsumer, except it will auto-encode and decode to JSON sent as WebSocket text frames.

Syntax-
```
from channels.generic.websocket import JsonWebsocketConsumer
class MyJsonWebsocketConsumer(JsonWebsocketConsumer):
    pass
```

- connect(self) - This handler is called when client initially opens a connection and is about to finish the WebSocket handshake.



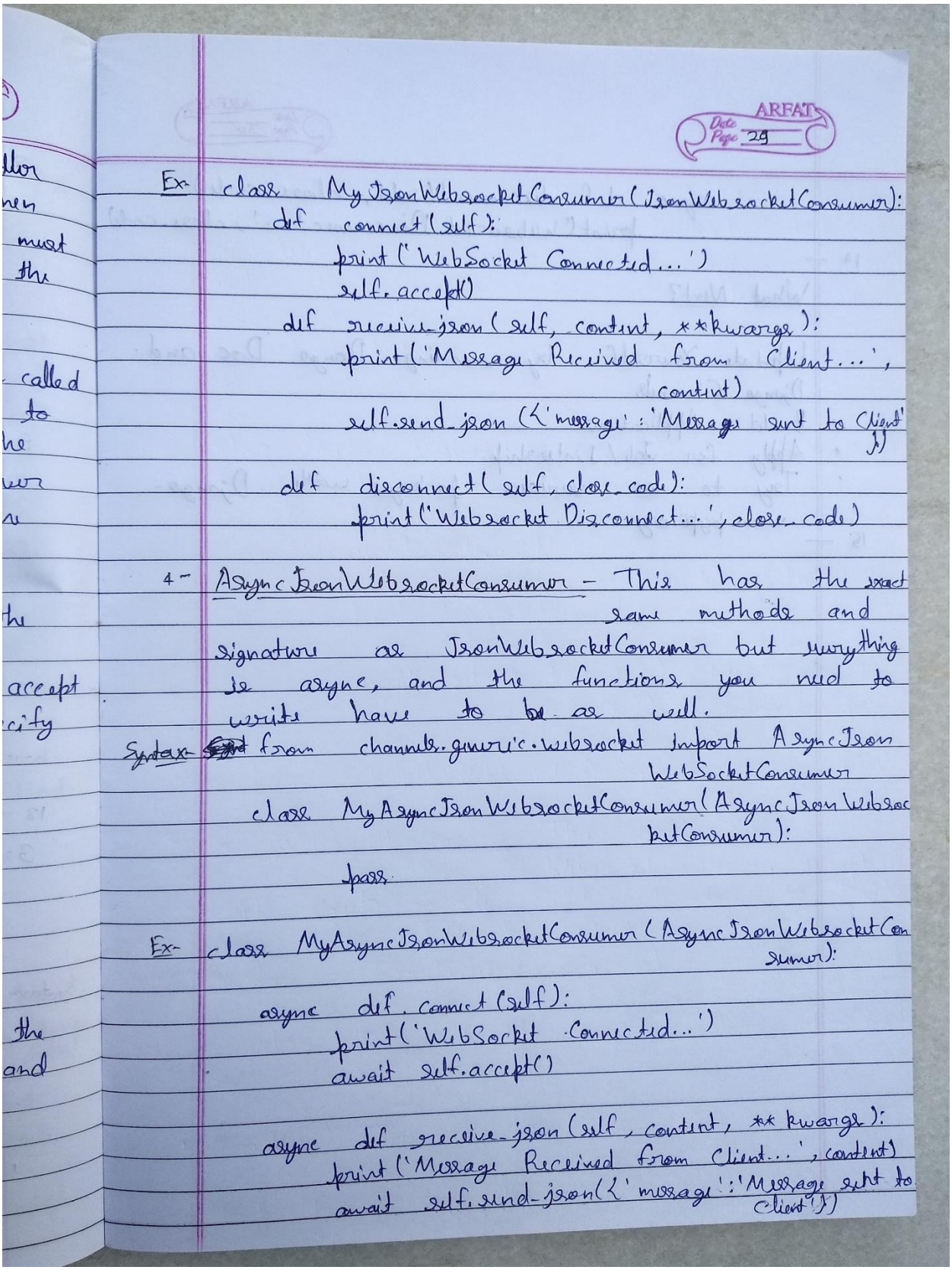
- receive_json(self, content, **kwargs) - This handler is called when data received from Client. This method must take a single argument, content, that is the decoded JSON object.

- disconnect(self, close_code) - This handler is called either connection to the client is lost, either from the client closing the connection, the server closing the connection, or loss of the socket.

- accept() - This is used to accept the connection.
- accept("subprotocol") - This is used to accept the connection and specify a chosen subprotocol.

- close() - This is used to reject the connection.
- close(code=4123) - This is used to reject connection with custom websocket error code.

- send_json(content) - This is used to Encode the given content as JSON and send it to the client.



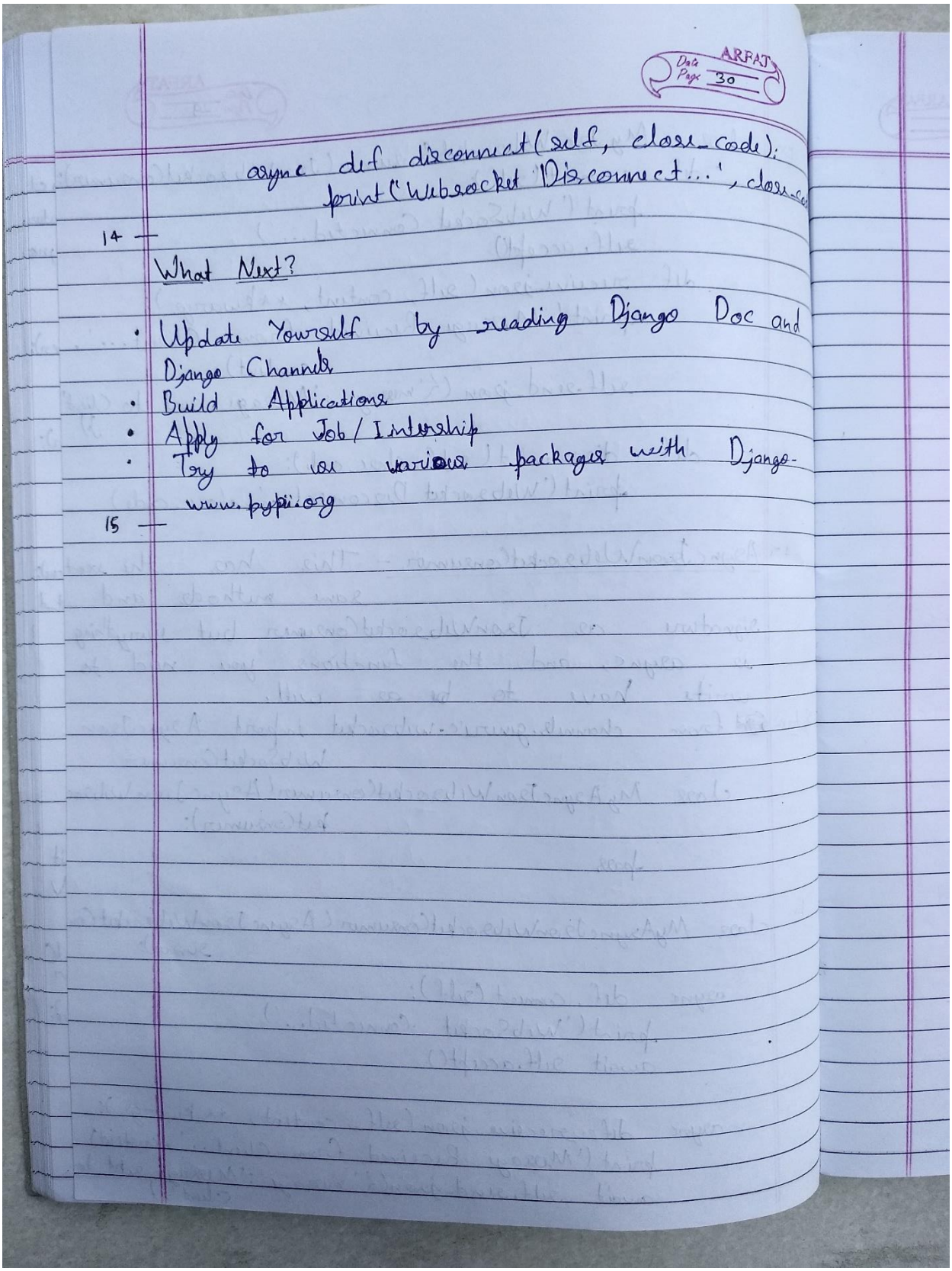
Ex- class MyJsonWebsocketConsumer(JsonWebsocketConsumer):
    def connect(self):
        print('WebSocket Connected...')
        self.accept()
    def receive_json(self, content, **kwargs):
        print('Message Received from Client...', content)
        self.send_json({'message': 'Message sent to Client'})
    def disconnect(self, close_code):
        print('Websocket Disconnect...', close_code)

4- AsyncJsonWebsocketConsumer - This has the exact same methods and signature as JsonWebsocketConsumer but everything is async, and the functions you need to write have to be as well.

Syntax- from channels.generic.websocket import AsyncJsonWebSocketConsumer

class MyAsyncJsonWebsocketConsumer(AsyncJsonWebsocketConsumer):
    pass

Ex- class MyAsyncJsonWebsocketConsumer(AsyncJsonWebsocketConsumer):
    async def connect(self):
        print('WebSocket Connected...')
        await self.accept()
    async def receive_json(self, content, **kwargs):
        print('Message Received from Client...', content)
        await self.send_json({'message': 'Message sent to Client'})



async def disconnect(self, close_code):
    print('Websocket Disconnect...', close_c

14 —

What Next?

- Update Yourself by reading Django Doc and Django Channels
- Build Applications
- Apply for Job / Internship
- Try to use various packages with Django- www.pypi.org

15 —