In endsem lab check for good keyboards then select the pc
Keep checking code line by line as you write

Diff syntax for diff questions so complete one ques then move on

>> is append > is subsitute

GIT

Always check git status before commits

wildcards
https://tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm

When you run the command

```bash
git add Documentation/\*.txt
```

the asterisk is prefixed by a backslash to prevent the shell from expanding it into a list of filenames before Git sees it [1]. By escaping the asterisk, the wildcard pattern is passed literally to Git, which then applies its own matching rules. This behavior is particularly useful because if the shell expanded the asterisk, it would only match files in the top level of the Documentation directory, potentially omitting files in its subdirectories. Passing the literal pattern allows Git to include files from subdirectories as intended [5].

Shell would expand and pass seperate filenames to git, but here the wildcard expression is the one being passed

---

BASH
—----------
Read x is like get line
$(( )) if we want instant substitution
(()) to just execute/update
(()) no problems with space and can refer variables without $

[[ ]] leave proper spaces

B1==$((x==y))
B1 gets 0 or 1 according to true/false

bc is a command-line utility, not some obscure part of shell syntax. The utility reads mathematical expressions from its standard input and prints values to its standard output. Since it is not part of the shell, it has no access to shell variables.

```
read exp

val=$(echo "scale=4; $exp" | bc -l)

if ((val > 0))
then
    echo $(echo "scale=3; $val + 0.0005" | bc -l)
fi
```

In line 5 if (()) bash cant handle(here cant compare) float so error

```
read exp

val=$(echo "scale=4; $exp" | bc -l)

if (($(echo "$val > 0" | bc -l) == 1))
then
    echo $(echo "scale=3; ($val + 0.0005)/1" | bc -l)
elif (($(echo "$val < 0" | bc -l) == 1))
then
    echo $(echo "scale=3; ($val - 0.0005)/1" | bc -l)
else
    echo $val
fi
```

```
x=$(cat)
echo 1 "$x"
echo 2 $x
```

Input (stdin)

```
Hello
World
how are you
```

Your Output (stdout)

```
1 Hello
World
how are you
2 Hello World how are you
```

“” activates the \n present in x in 2 they are compressed to spaces

```
# x=$(cat)
# echo 1 "$x"
# echo 2 $x

while read line; do
 #Reading each line
 echo $(echo "$line" | cut -b 3)
 done

```

read line without any file like cat without file but stops after one empty line

```
while read line
do
    echo $(echo "$line" | cut -b 2,7)
done
```

should be cut -c (-b not always work)

Input (stdin)

```
1 hello
2
3 hello
```

Your Output (stdout)

```
1 e
2
```

Default delim for cut is tab
Is no delim in a line prints the complete line

PYTHON—----------

```python
class test:
    def __init__(self, relations = []):
        self.relations = relations
        print(id(self.relations))

def func(x = 1):
    print(id(x))

a = test() # same
b = test()
func() # same
func()
print(id(a), id(b)) # different
a = [] # both point to different objects
b = []
```

print(f”{x:.1f}”) 1 decimal

- While reading form file make sure to convert everything to int because it will be string

Reshape not modify the memory

AWK—----------

Default print

Unlike sed which prints the outputs(??) by default
OFS, ORS work fo print
Awk takes a line(identified by RS) then breaks into fields by FS then does processing no OFS ORS involved unless we use print

Print $1, $2
Spaces don't matter 0 spaces will be between the two fields just OFS

OFS ORS don't act on printf