Mr. C loves Round Trips

ESC101: Fundamentals of Computing

Purushottam Kar

Announcements - Quiz

- Major quiz **tomorrow** – (syllabus till **Friday Aug 24**)
  - Wednesday, August 29, 2018, 12PM-12:50PM, L20 (i.e. lecture hour)
  - During lecture hours – don't be absent
  - **Bring your institute ID card** with you – will lose time if you forget
  - No minor quizzes during lab this week (August 27-August 30)
- Bring a **pencil, eraser and sharpener** with you
  - Answers to be written on question paper itself and returned back
  - If you make a mistake with pen – no extra question papers
  - If unsure, **first write answer with pencil** and **finally write it in pen**
  - We WONT HAVE EXTRA QUESTION PAPERS in case you spoil yours
  - We WONT HAVE PENCILS, ERASERS in case you forget

Announcements - Holiday

- Institute holiday next Monday (03 September, 2018)
- No lecture, no lab on that day
- Extra lecture on Saturday 08 September, 2018
  - 12 noon, L20 (same as usual)
  - Scheduled by DoAA, not by me – I like to sleep on Sat too ☹
- Extra lab for B1, B2, B3 on Saturday 08 September
  - 2PM – 5PM, New Core Labs CC-02 (same as usual)

The for loop

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

The for loop

General form of a for loop

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

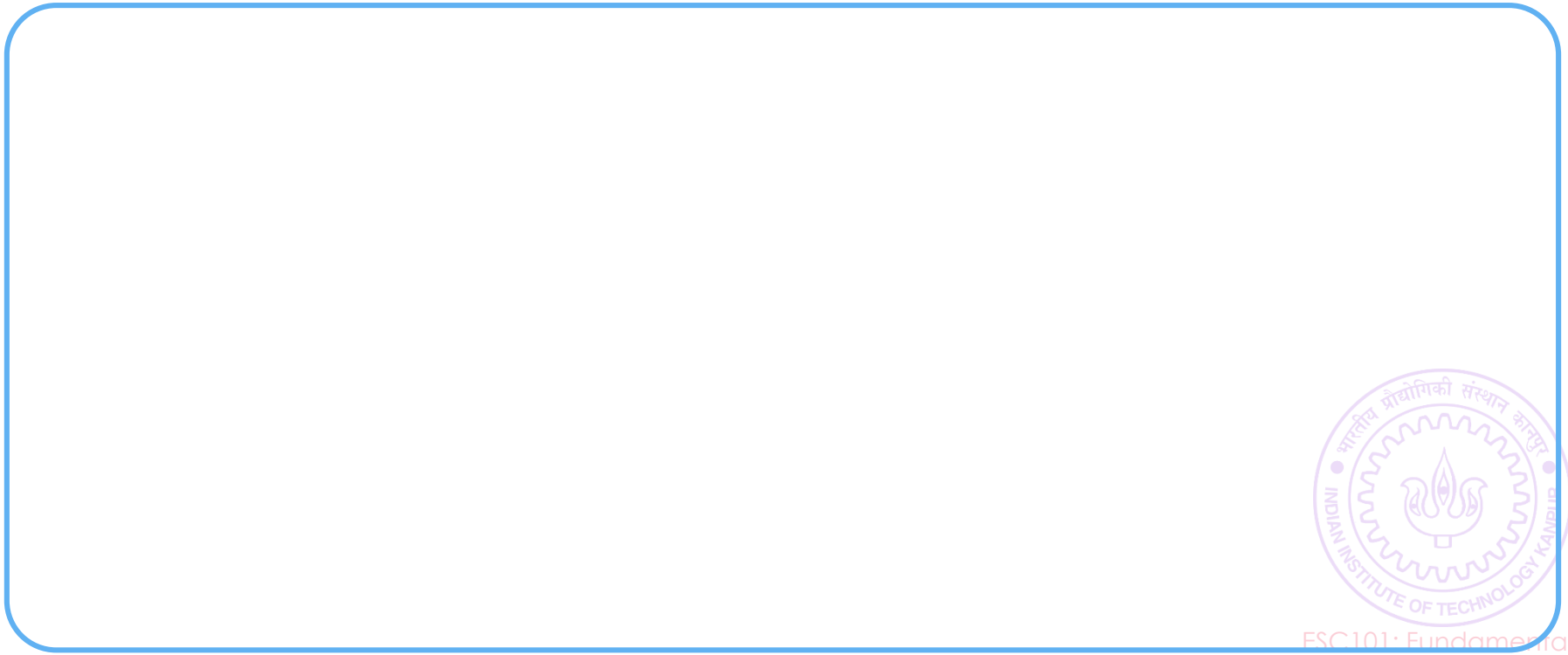
How we usually speak to a human

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human



The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   Execute all statements inside braces

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   Execute all statements inside braces

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression

The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2

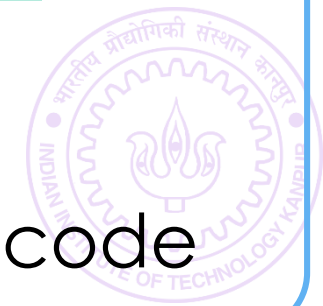
The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2
   - Else stop looping and execute rest of code



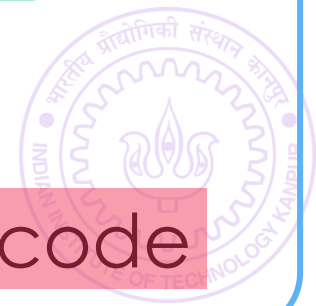
The for loop

General form of a for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2
   - Else stop looping and execute rest of code



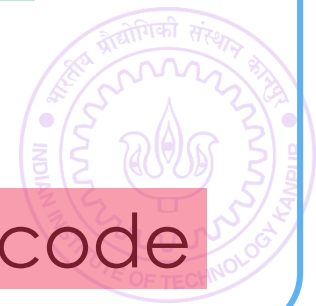
The for loop

General form of a for loop

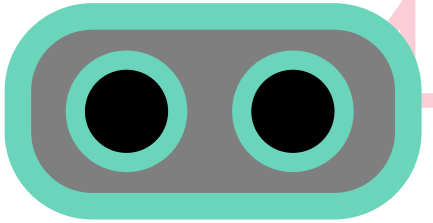
```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2
   - Else stop looping and execute rest of code



The for loop



General form of a for loop

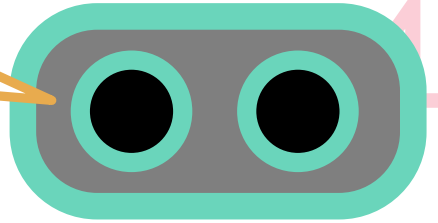
```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2
   - Else stop looping and execute rest of code

The for loop

Brackets essential if you want me to do many things while looping



General form of a for loop

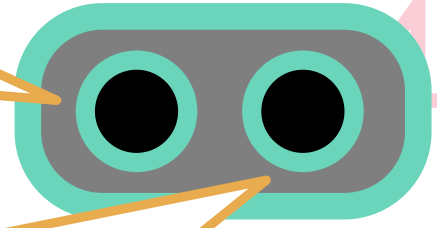
```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2
   - Else stop looping and execute rest of code

The for loop

Brackets essential if you want me to do many things while looping



Each time I execute the statements inside the braces – called one *iteration* of the loop

General form of a for loop

```
for(init_expr; stopping_expr; update_
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2
   - Else stop looping and execute rest of code

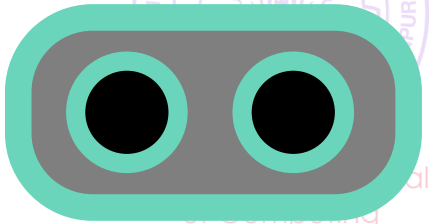
Some useful Tips for using Loops

Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again

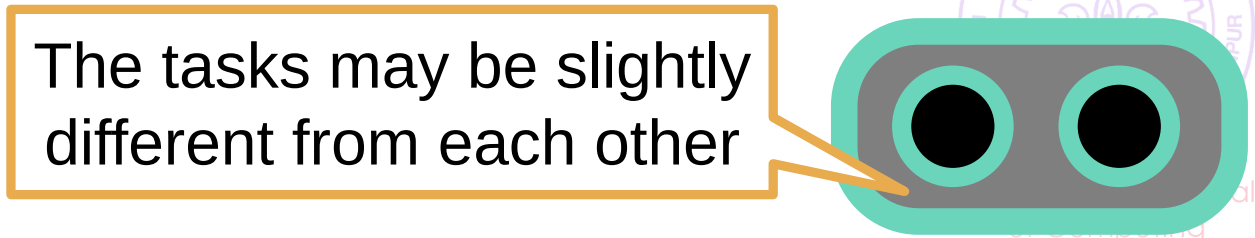
Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again



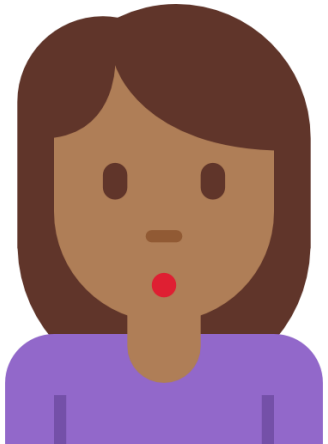
Some useful Tips for using Loops

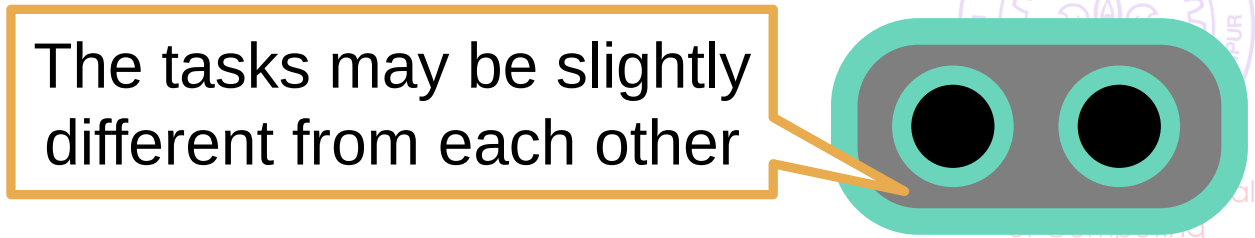
Read the problem carefully and identify some tasks that have to be repeated again and again



Some useful Tips for using Loops

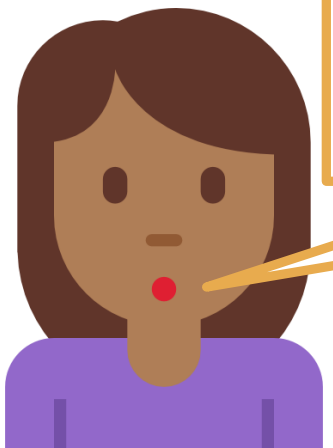
Read the problem carefully and identify some tasks that have to be repeated again and again





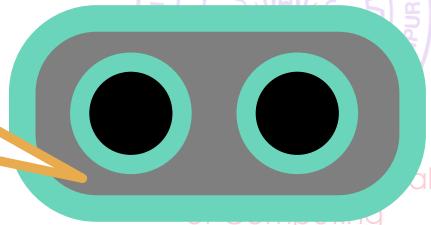
Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again



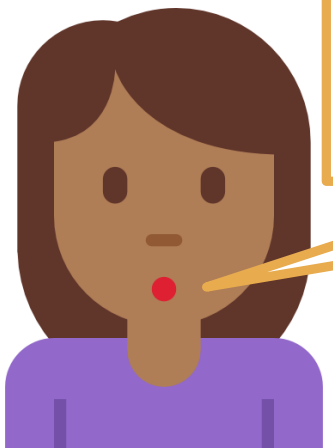
Yes, in the multiplication table example, the tasks were slightly different. First print 2 x 1 = 2, then print 2 x 2 = 4 etc etc.

The tasks may be slightly different from each other



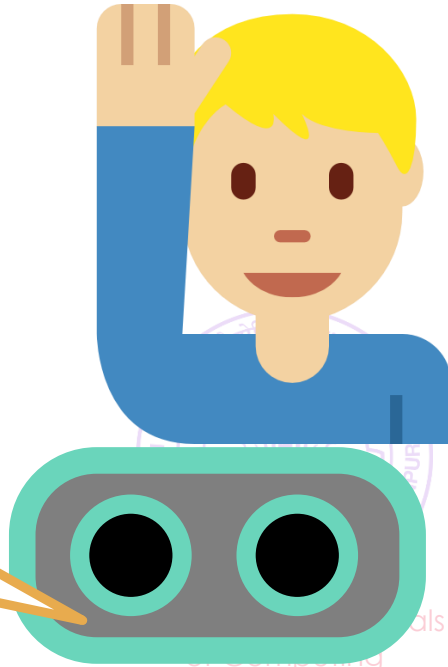
Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again



Yes, in the multiplication table example, the tasks were slightly different. First print 2 x 1 = 2, then print 2 x 2 = 4 etc etc.

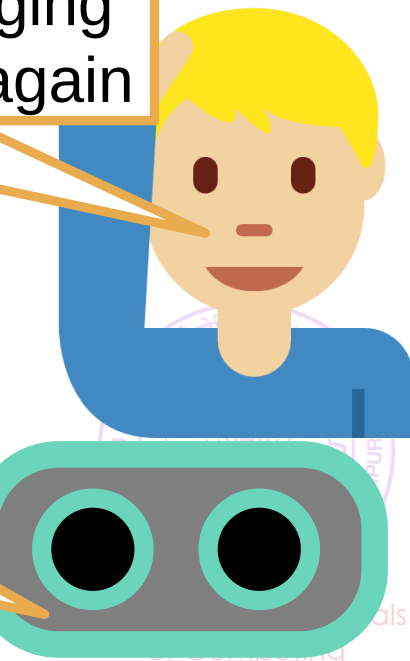
The tasks may be slightly different from each other



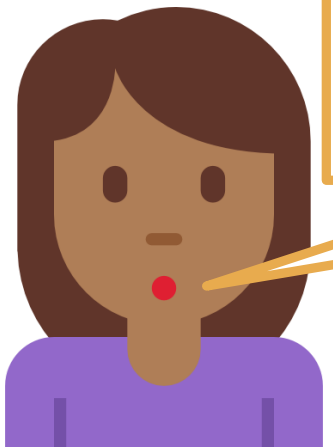
Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again

Yes, but we could write the same code
printf("%d x %d = %d\n", a, b, a*b);
to do all the tasks by simply changing the value of variable b again and again



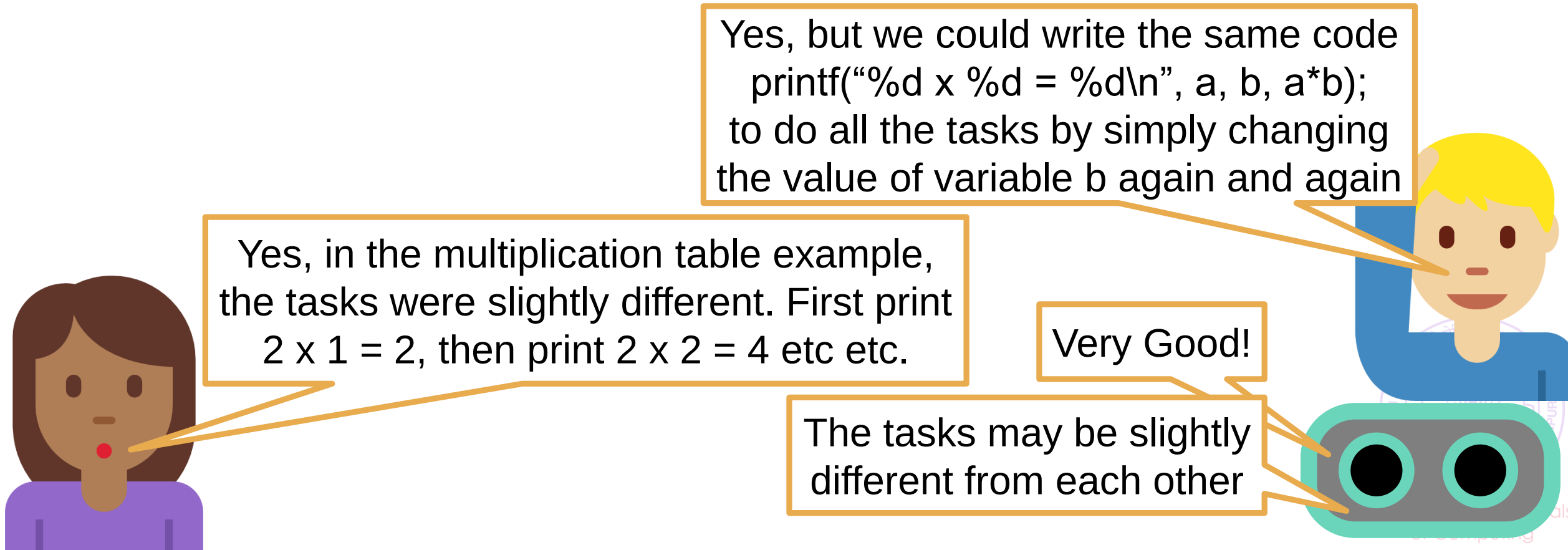
Yes, in the multiplication table example, the tasks were slightly different. First print 2 x 1 = 2, then print 2 x 2 = 4 etc etc.



The tasks may be slightly different from each other

Some useful Tips for using Loops

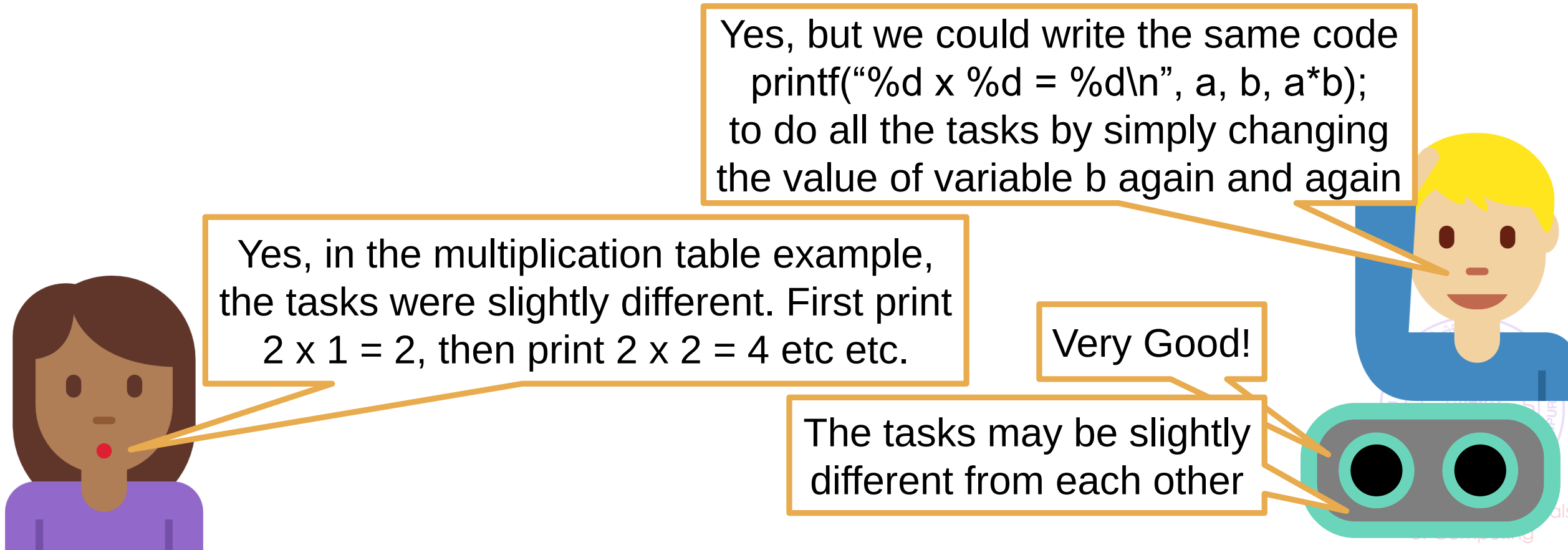
Read the problem carefully and identify some tasks that have to be repeated again and again



Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again

Use this variable that is changing as the variable of loop



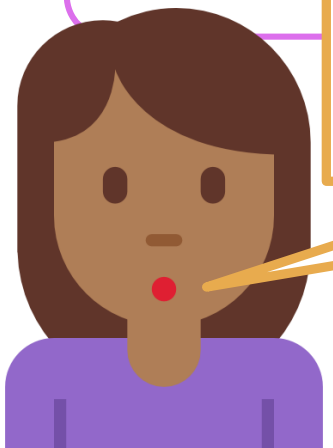
Some useful Tips for using Loops

Read the problem carefully and identify some tasks that have to be repeated again and again

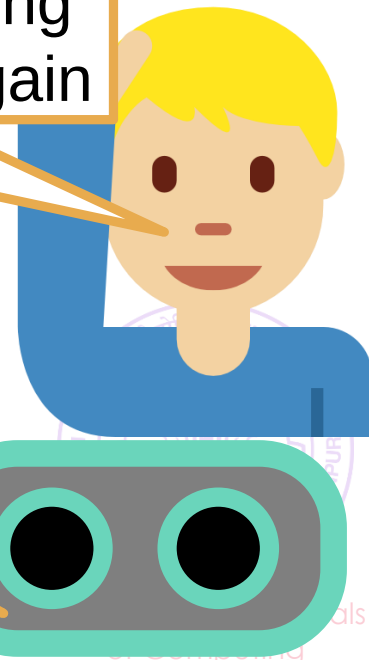
Use this variable that is changing as the variable of loop

```
int a = 2, b;
for(b = 1; b <= 10; b++){
    printf("%d x %d = %d\n", a, b, a*b);
}
```

Yes, but we could write the same code printf("%d x %d = %d\n", a, b, a*b); to do all the tasks by simply changing the value of variable b again and again



Yes, in the multiplication table example, the tasks were slightly different. First print 2 x 1 = 2, then print 2 x 2 = 4 etc etc.



Very Good!

The tasks may be slightly different from each other

When loops are not very useful

When loops are not very useful

Read a number then print a number, then read another number and then print it – repeat 100 times

When loops are not very useful

Read a number then print a number, then read another number and then print it – repeat 100 times

- Loops are very very useful for problems like the above

When loops are not very useful

Read a number then print a number, then read another number and then print it – repeat 100 times

- Loops are very very useful for problems like the above

Read a number then print it, then calculate the 4^{th} power of 20, then print the string "Hello World", then read another number, then add it to the first number, then print the string "Bye" and then calculate log(first number), then calculate sin(second number)

When loops are not very useful

Read a number then print a number, then read another number and then print it – repeat 100 times

- Loops are very very useful for problems like the above

Read a number then print it, then calculate the 4^{th} power of 20, then print the string "Hello World", then read another number, then add it to the first number, then print the string "Bye" and then calculate log(first number), then calculate sin(second number)

- Long list of tasks but no structure, no repeated pattern

When loops are not very useful

Read a number then print a number, then read another number and then print it – repeat 100 times

- Loops are very very useful for problems like the above

Read a number then print it, then calculate the 4^{th} power of 20, then print the string "Hello World", then read another number, then add it to the first number, then print the string "Bye" and then calculate log(first number), then calculate sin(second number)

- Long list of tasks but no structure, no repeated pattern
- Loops are not very useful for the above problem

Print sum of reciprocals of 1, 2 … n

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The repeating task can be

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Define a variable (lets call it **sum**) to store partial sums

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Define a variable (lets call it **sum**) to store partial sums

Lovingly called *partial sums* or *running sums*

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Define a variable (lets call it **sum**) to store partial sums

The above task is accomplished by the code

Lovingly called *partial sums* or *running sums*

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

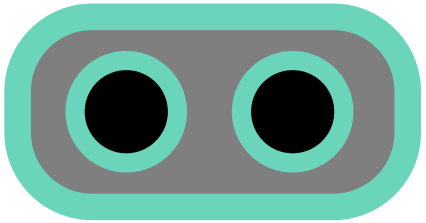
Define a variable (lets call it **sum**) to store partial sums

The above task is accomplished by the code

```
sum = sum + 1/i;
```

Lovingly called *partial sums* or *running sums*

Print sum of reciprocals of 1, 2 … n



Take n >= 1 from the user and give as output

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$$

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

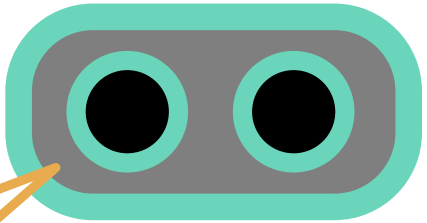
Define a variable (lets call it **sum**) to store partial sums

The above task is accomplished by the code

```
sum = sum + 1/i;
```

Lovingly called *partial sums* or *running sums*

Print sum of reciprocals of 1, 2 … n



Take n >= 1 from the user and give as output

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

Oops! Integer division!

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Define a variable (lets call it **sum**) to store partial sums

The above task is accomplished by the code

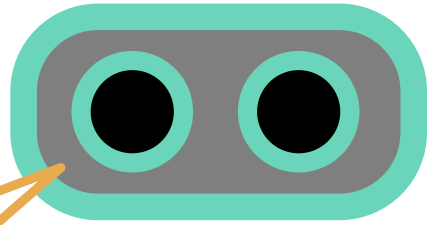
```
sum = sum + 1/i;
```

Lovingly called *partial sums* or *running sums*

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$



Oops! Integer division!

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Define a variable (lets call it **sum**) to store partial sums

The above task is accomplished by the code

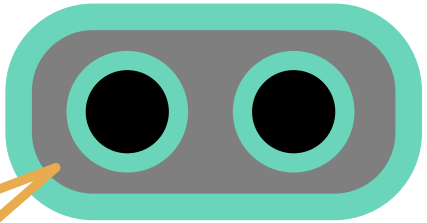
```
sum = sum + 1.0/i;
```

Lovingly called *partial sums* or *running sums*

Print sum of reciprocals of 1, 2 … n

Take n >= 1 from the user and give as output

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$



Oops! Integer division!

The repeating task can be

Given the sum of first i-1 reciprocals and add $\frac{1}{i}$ *to it*

Define a variable (lets call it **sum**) to store partial sums

The above task is accomplished by the code

```
sum = sum + 1.0/i;
```

```
sum = sum + (double)1/i;
```

Lovingly called *partial sums* or *running sums*

Loop Invariants

Loop Invariants

Notice that in previous example

Loop Invariants

Notice that in previous example

At the beginning of i-th iteration, sum stored the value

Loop Invariants

Notice that in previous example

At the beginning of i-th iteration, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i-1}$$

Loop Invariants

Notice that in previous example

At the beginning of i-th iteration, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i-1}$$

Except for the special case for the iteration with i = 1, sum stored 0

Loop Invariants

Notice that in previous example

At the beginning of i-th iteration, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i-1}$$

Except for the special case for the iteration with i = 1, sum stored 0

After the i-th iteration is over, sum stored the value

Loop Invariants

Notice that in previous example

At the beginning of i-th iteration, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i-1}$$

Except for the special case for the iteration with i = 1, sum stored 0

After the i-th iteration is over, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i}$$

Loop Invariants

Notice that in previous example

At the beginning of i-th iteration, sum stored the value

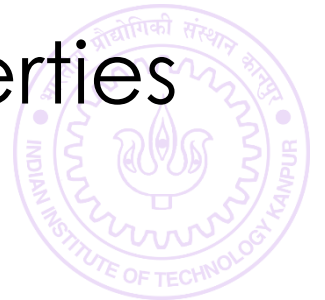
$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i-1}$$

Except for the special case for the iteration with i = 1, sum stored 0

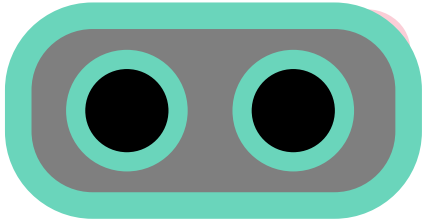
After the i-th iteration is over, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop



Loop Invariants



Notice that in previous example

At the beginning of i-th iteration, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i-1}$$

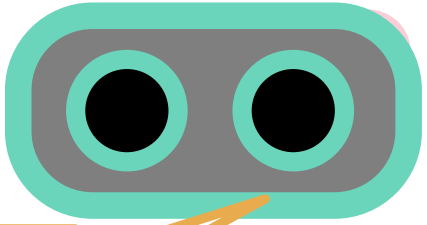
Except for the special case for the iteration with i = 1, sum stored 0

After the i-th iteration is over, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Loop Invariants



Notice that in previous e

Loop invariants are powerful ways to ensure that your loop code is correct!

At the beginning of i-th iteratio

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i-1}$$

Except for the special case for the iteration with i = 1, sum stored 0

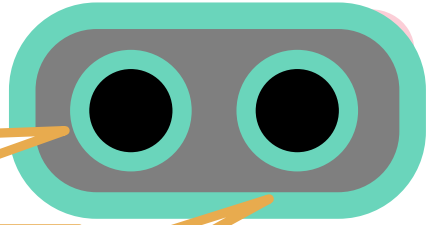
After the i-th iteration is over, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Loop Invariants

Very important once loops get more complicated



Loop invariants are powerful ways to ensure that your loop code is correct!

Notice that in previous e

At the beginning of i-th iteratio

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i-1}$$

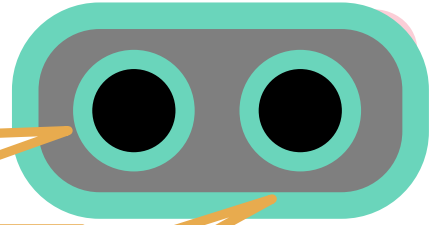
Except for the special case for the iteration with i = 1, sum stored 0

After the i-th iteration is over, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Loop Invariants



Very important once loops get more complicated

Loop invariants are powerful ways to ensure that your loop code is correct!

Notice that in previous e

At the beginning of i-th iteratio

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i-1}$$

Except for the special case for the iteration with i = 1, sum stored 0

After the i-th iteration is over, sum stored the value

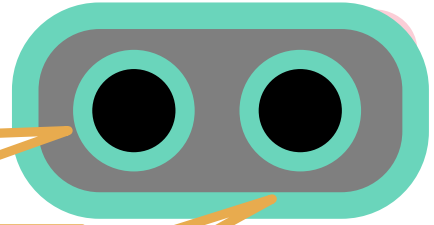
$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Quiz: what is loop invariant in multiplication table code?

Loop Invariants

Very important once loops get more complicated

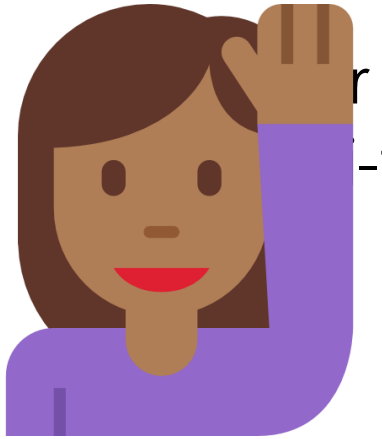


Loop invariants are powerful ways to ensure that your loop code is correct!

Notice that in previous e

At the beginning of i-th iteratio

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{i-1}$$



r the special case for the iteration with i = 1, sum stored 0

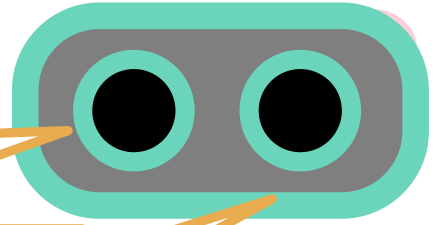
-th iteration is over, sum stored the value

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Quiz: what is loop invariant in multiplication table code?

Loop Invariants



Very important once loops get more complicated

Loop invariants are powerful ways to ensure that your loop code is correct!

Notice that in previous e

At the beginning of i-th iteratio

$$\frac{1}{-} + \frac{1}{-} + \frac{1}{3} + \ldots + \frac{1}{i-1}$$

In i-th iteration the string 2 x i = 2i will get printed

e iteration with i = 1, sum stored 0

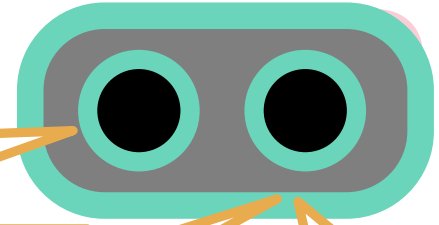
iteration is over, sum stored the value

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Quiz: what is loop invariant in multiplication table code?

Loop Invariants



Very important once loops get more complicated

Loop invariants are powerful ways to ensure that your loop code is correct!

Very Good!

Notice that in previous e

At the beginning of i-th iteratio

$$\frac{1}{-} + \frac{1}{-} + \frac{1}{3} + \ldots + \frac{1}{i-1}$$

In i-th iteration the string 2 x i = 2i will get printed

e iteration with i = 1, sum stored 0

iteration is over, sum stored the value

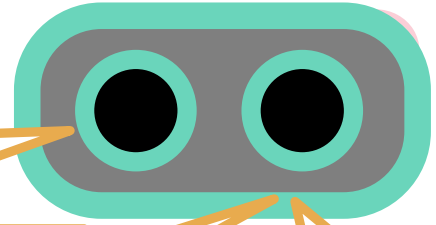
$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

These are known as *loop invariants* – few nice properties that hold for all iterations of a loop

Quiz: what is loop invariant in multiplication table code?

Loop Invariants

Very important once loops get more complicated



Loop invariants are powerful ways to ensure that your loop code is correct!

Very Good!

Notice that in previous e

At the beginning of i-th iteratio

$$\frac{1}{-} + \frac{1}{-} + \frac{1}{3} + \ldots + \frac{1}{i-1}$$

In i-th iteration the string 2 x i = 2i will get printed

e iteration with i = 1, sum stored 0

iteration is over, sum stored the value

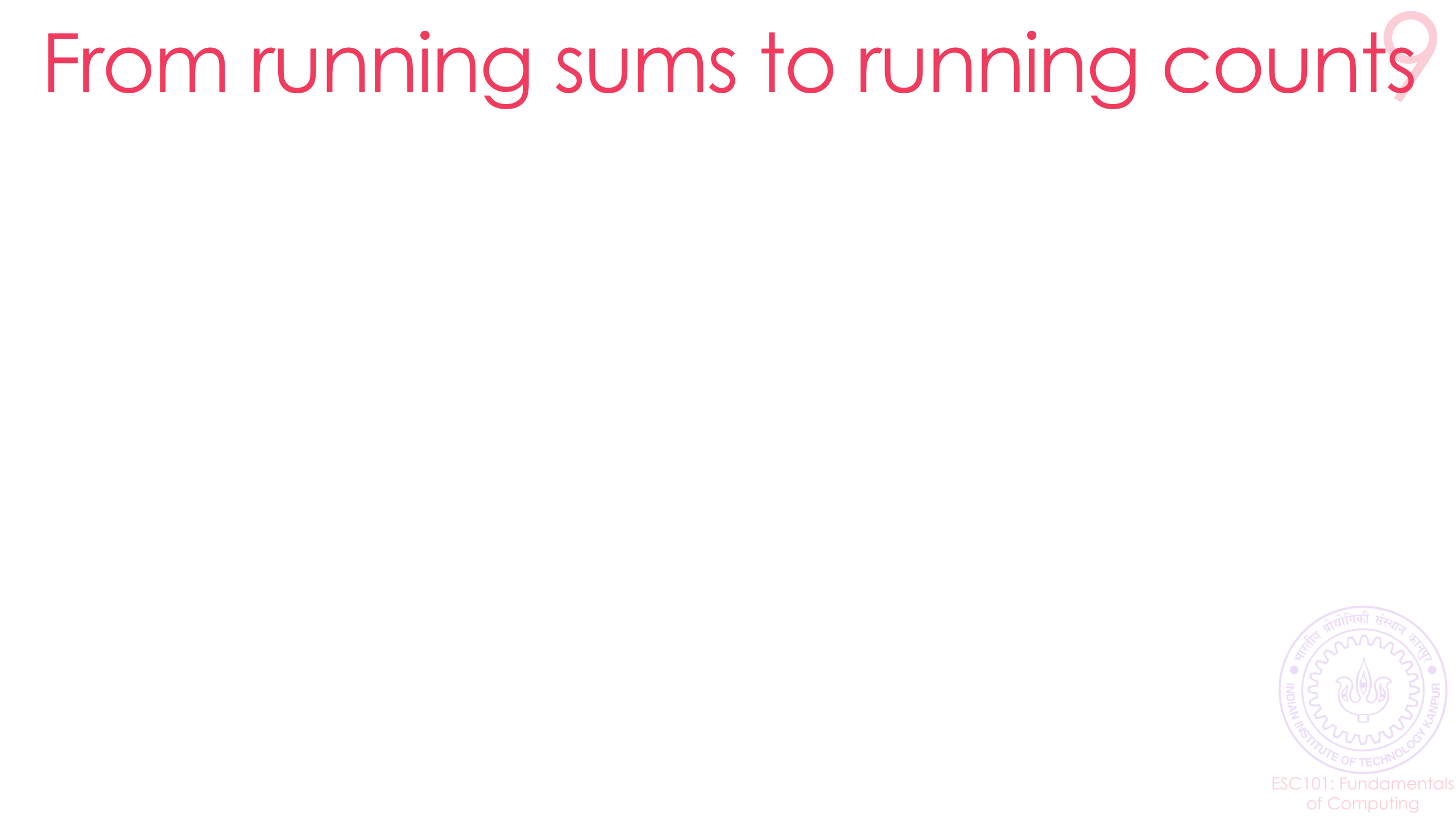
$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

These are known as *loop invariants*, a few nice properties that hold for all i

Quiz: what is loop invariant in multiplication table code?

Exercise 1: sum of reciprocals of the first n even numbers
Exercise 2*: sum of reciprocals of the first n prime numbers

From running sums to running counts



From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

Repeating task seems to be

From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

Repeating task seems to be

Given the count of numbers div by 3 or 5 till now, read another number and add 1 to count if div by 3 or 5

From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

Repeating task seems to be

Given the count of numbers div by 3 or 5 till now, read another number and add 1 to count if div by 3 or 5

Here, we can use a variable (lets call it **count**) to store the *running count* of numbers div by 3 or 5 seen till now

From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

Repeating task seems to be

Given the count of numbers div by 3 or 5 till now, read another number and add 1 to count if div by 3 or 5

Here, we can use a variable (lets call it **count**) to store the *running count* of numbers div by 3 or 5 seen till now

Loop invariants

From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

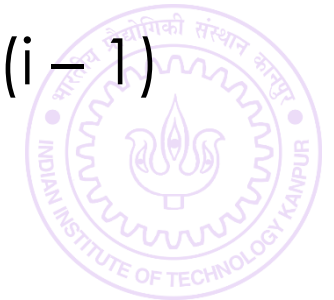
Repeating task seems to be

Given the count of numbers div by 3 or 5 till now, read another number and add 1 to count if div by 3 or 5

Here, we can use a variable (lets call it **count**) to store the *running count* of numbers div by 3 or 5 seen till now

Loop invariants

At the beginning of i-th iteration, count stores how many of the first (i – 1) numbers were divisible by 3 or divisible by 5



From running sums to running counts

Read n from input, then scan n numbers, and print how many numbers were divisible by 3 or divisible by 5

Repeating task seems to be

Given the count of numbers div by 3 or 5 till now, read another number and add 1 to count if div by 3 or 5

Here, we can use a variable (lets call it **count**) to store the *running count* of numbers div by 3 or 5 seen till now

Loop invariants

- At the beginning of i-th iteration, count stores how many of the first (i – 1) numbers were divisible by 3 or divisible by 5
- After the i-th iteration is over, count stores how many of the first i numbers were divisible by 3 or divisible by 5

The while loop

The while loop

Remember that we could write fancy for loops where init_expr and update_expr were empty?

The while loop

Remember that we could write fancy for loops where init_expr and update_expr were empty?

```
for(;stopping_expr;){ ... }
```

The while loop

Remember that we could write fancy for loops where init_expr and update_expr were empty?

```
for(;stopping_expr;){ ... }
```

If we do the above, we had to write the init_expr before starting the for loop and write update_expr inside loop

The while loop

Remember that we could write fancy for loops where init_expr and update_expr were empty?

```
for(;stopping_expr;){ ... }
```

If we do the above, we had to write the init_expr before starting the for loop and write update_expr inside loop

There is a much neater way to write the above

The while loop

Remember that we could write fancy for loops where init_expr and update_expr were empty?

```
for(;stopping_expr;){ ... }
```

If we do the above, we had to write the init_expr before starting the for loop and write update_expr inside loop

There is a much neater way to write the above

```
while(stopping_expr){
    statement1;
    statement2;
}
```

The while loop

The while loop

General form of a while loop

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

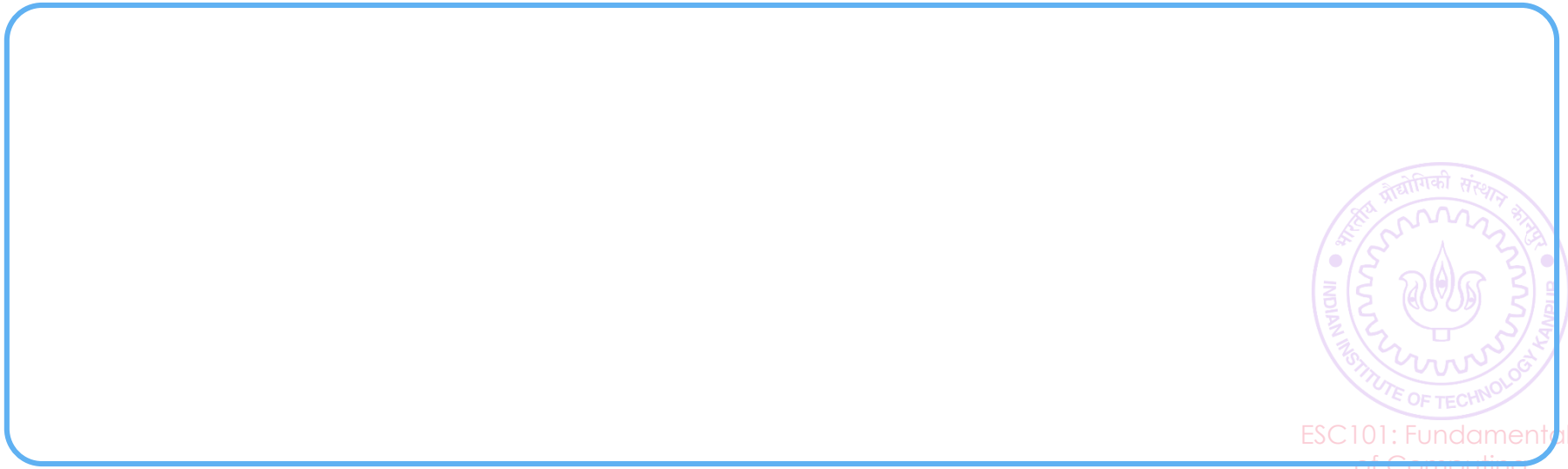
How we usually speak to a human

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human



The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   Execute all statements inside braces

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   Execute all statements inside braces

The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   Execute all statements inside braces
   Go back to step 2

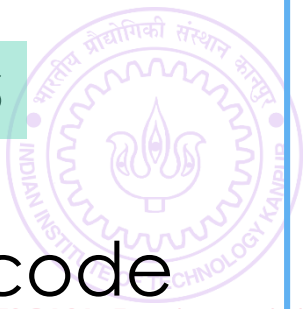
The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code



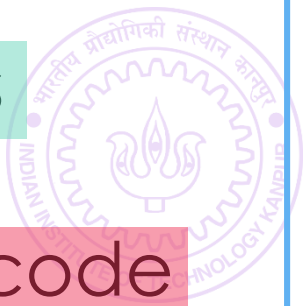
The while loop

General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code



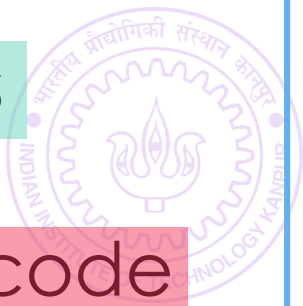
The while loop

General form of a while loop

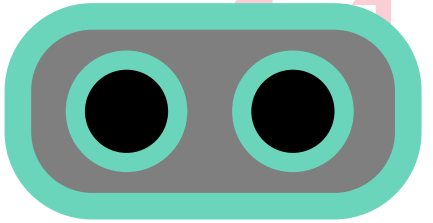
```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code



The while loop



General form of a while loop

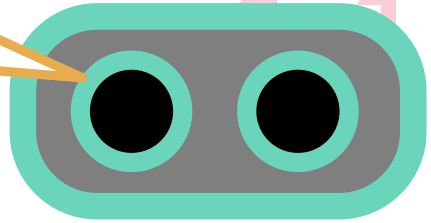
```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code

The while loop

Brackets essential if you want me to do many things while looping



General form of a while loop

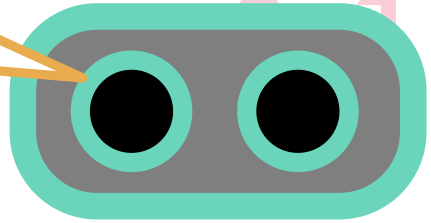
```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code

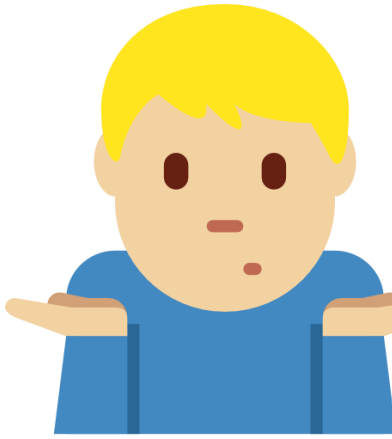
The while loop

Brackets essential if you want me to do many things while looping



General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

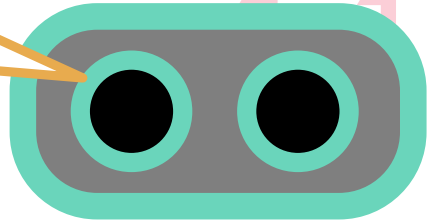


How we usually speak to a human

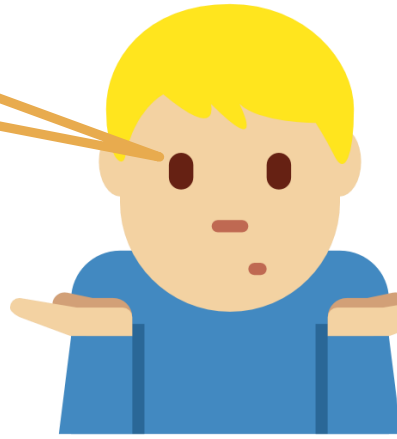
1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code

The while loop

Brackets essential if you want me to do many things while looping



So what is the difference between for and while?



General form of a while loop

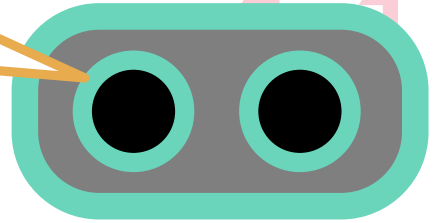
```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

How we usually speak to a human

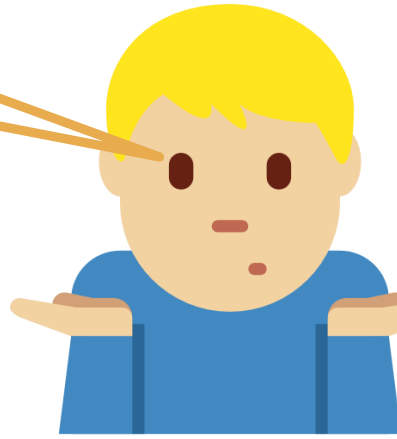
1. First check the stopping expression
2. If stopping expression is true
   Execute all statements inside braces
   Go back to step 2
   Else stop looping and execute rest of code

The while loop

Brackets essential if you want me to do many things while looping

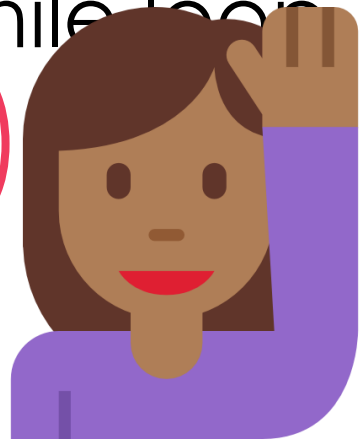


So what is the difference between for and while?



General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```



How we usually speak to a human

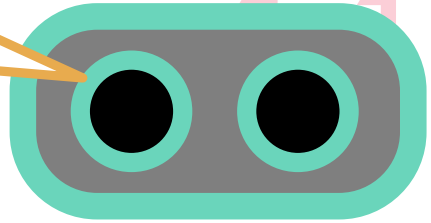
1. First check the stopping expression
2. If stopping expression is true
   - Execute all statements inside braces
   - Go back to step 2
   - Else stop looping and execute rest of code

The while loop

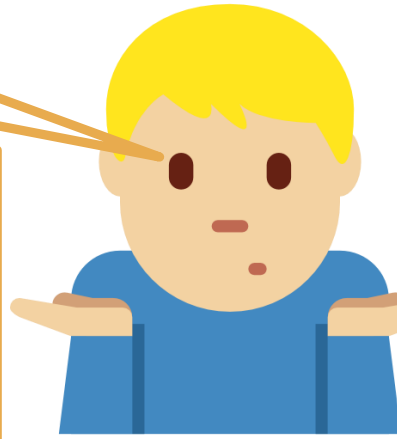
General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

Brackets essential if you want me to do many things while looping



So what is the difference between for and while?



In general not much – it is a matter of style. Use while when you don't know how many iterations will loop run

How we usually speak to a human

1. First check the stopping expression
2. If stopping expression is true
   Execute all statements inside braces
   Go back to step 2
   Else stop looping and execute rest of code

Scanning a list of numbers

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it **sum**) to store partial sums

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it `sum`) to store partial sums

Even while loops have invariants

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it **sum**) to store partial sums

Even while loops have invariants

- At the beginning of i-th iteration, sum will store sum of first (i – 1) numbers

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it **sum**) to store partial sums

Even while loops have invariants

- At the beginning of i-th iteration, sum will store sum of first (i – 1) numbers
- After the i-th iteration is over, sum will store sum of first i numbers

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it **sum**) to store partial sums

Even while loops have invariants

- At the beginning of i-th iteration, sum will store sum of first (i – 1) numbers
- After the i-th iteration is over, sum will store sum of first i numbers

However, there is no clear variable of the loop here

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it **sum**) to store partial sums

Even while loops have invariants

- At the beginning of i-th iteration, sum will store sum of first (i – 1) numbers
- After the i-th iteration is over, sum will store sum of first i numbers

However, there is no clear variable of the loop here

Usually if you can find a nice clear variable of loop, try using for loops

Scanning a list of numbers

Read integers from the input till you read the number -1 and print the sum of all numbers except the -1

Here, number of iterations is not given to us as it was in the divisible-by-3-or-5 or reciprocal sum examples

Use a variable (lets call it **sum**) to store partial sums

Even while loops have invariants

- At the beginning of i-th iteration, sum will store sum of first (i – 1) numbers
- After the i-th iteration is over, sum will store sum of first i numbers

However, there is no clear variable of the loop here

Usually if you
using for loo

Exercise 1: read integers till -1, print sum of all primes in the list
Exercise 2*: read integers till -1, print sum of all numbers greater than the number encountered just before that number