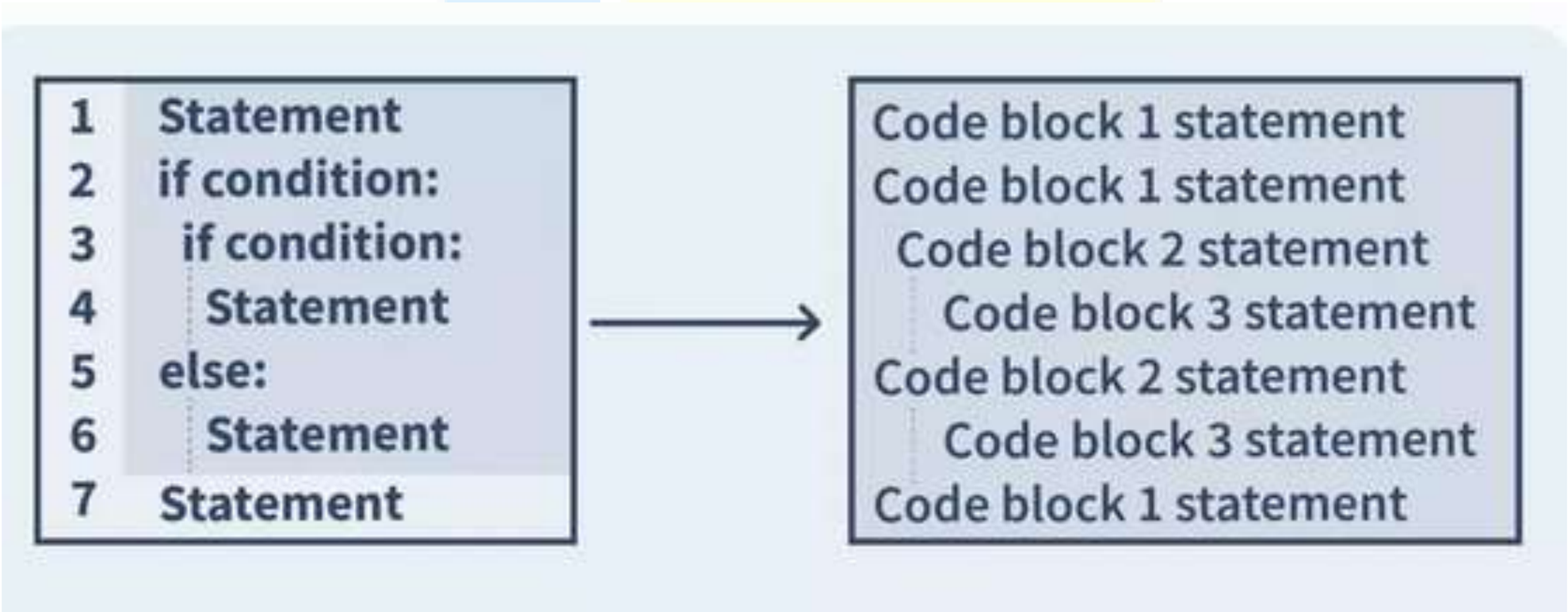
Apply Python

LU3 – Develop Python Application Part 2

Eng. NDAYISENGA Gilbert
UNIVERSITY OF KIGALI

Indentation in Python

Indentation is the leading whitespace (*spaces or/and tabs*) before any statement in Python. The reason why indentation is important in python is that the indentation serves another purpose other than code readability. Python treats the statements with the same indentation level (statements with an equal number of whitespaces before them) as a single code block. So whereas in languages like C, C++, etc. a block of code is represented by curly braces { }, in python a block is a group of statements that have the same Indentation level i.e same number of leading whitespaces.



Below are some of the observations that can be made from the above figure:

- All the statements which are on the same level of Indentation (Same no of whitespaces before them) belong to a single block, so from the above diagram, statements in line 1, line 2, and line 7 belong to a single block, and the block has the zero or lowest level of indentation. Statements 3 and 5 are indented one step, forming another block at the first level of indentation. Similarly, statements 4 and 6 are indented two steps, so they together form another block at the second level of indentation.
- Below the line 2 statement, which is an if statement, statements 3 and 5 are indented one step; hence, they belong to a single block. And since line 2 is an if statement, the block indented below the first if forms the body of second if. So here, the body of the if statement at line 2 includes all the lines that are indented below it, i.e., lines 3,4,5 and 6.
- Now that we know that statement at line numbers 3,4,5 and 6 forms the body of the if statement at line 2. Let us understand the indentation for them. Statements at 3 and 5 are uniformly indented, so they belong to a single block (block2 from the interpretation), and they will be executed one by one.
- Statement at line 4 makes up the body of the if statement at line 3, as we know any statements that are indented below an if form the body of if statement, similarity the statement at line 6 makes up the body of else statement at line 5.
- This is how the indentation helps define the blocks and also to identify to which statements the block belongs.

Execution

- The execution starts at line 1 followed by the statement at line 2; if the condition is evaluated and in case it returns true, then control goes inside the body of the if statement, which brings statements 3,4, 5, and 6 to the picture.
  - Now, the statement at line 3 is executed, and if the condition is evaluated, in case it returns true, then line 4 is executed, after which control goes to line 7. If the condition at line 3 returns false, then control goes to another statement that is line 5, and then line 6 is executed, followed by the statement at line 7.
- In case condition at line number 2 returns false, the control skips lines 3, 4, 5, and 6 and goes to the statement at line 7.

Example: Below is an example code snippet with the correct indentation in python.

Code:

```
name = 'Gilbert'
if name == 'Gilbert':
   print('Welcome Gilbert..')
   print('How are you?')
else:
   print('Dude! whoever you are ')
   print('Why are you here?')

print('Have a great day!')
```

Output:

```
Welcome Gilbert..
How are you?
Have a great day!
```

Explanation:

- The name variable gets assigned to Rahul in the first statement
- Now the statement if name == 'Gilbert': is evaluated, it returns true, so it executes the body of the if, which is the indented next two statements below the if statement. The two statements inside the body are print('Welcome Gilbert..') and print('How are you?') and they get executed.
- Once the statement gets executed the else part is skipped and control goes to the next statement which is print('Have a great day!'), which is executed.
- In this program, the statements inside the bodies of "if" and "else" are indented.

Python Indentation Rules

- Python uses **four spaces as default** indentation spaces. However, the number of spaces can be anything; it is up to the user. But a **minimum of one space** is needed to indent a statement.
- The first line of python code cannot have an indentation.
- Indentation is mandatory in python to define the blocks of statements.
- The number of spaces must be uniform in a block of code.

- It is **preferred to use whitespaces** instead of tabs to indent in python. Also, either use whitespace or tabs to indent; intermixing of tabs and whitespaces in indentation can cause wrong indentation errors.

Benefits of Indentation in Python

- Indentation of code leads to better readability, although the primary reason for indentation in python is to identify block structures.
- Missing {and} errors that sometimes popup in C, C++ languages can be avoided in python; also, the number of lines of code is reduced.

Flow Control in Python?

There are two words, the first is flow and the second is control. Let's try to understand them and make a relationship out of it.

From a technical perspective, we can say that flow is nothing but many statements consisting of multiple operations which need to be executed. For example, add 2 and 3 is a statement (or expression in this case) where operation is addition. Like this, we can have similar or different kinds of statements according to the problem we're solving through the code.

Now, we have got the statements that we include in our code. But the thing we're missing is the control. We don't have any control over the statements yet. By control, I mean the decision power. To relate this with real life, one can imagine that in their day-to-day life, they make decisions whether to do this or that, choose this or that, etc. And, after deciding we follow the

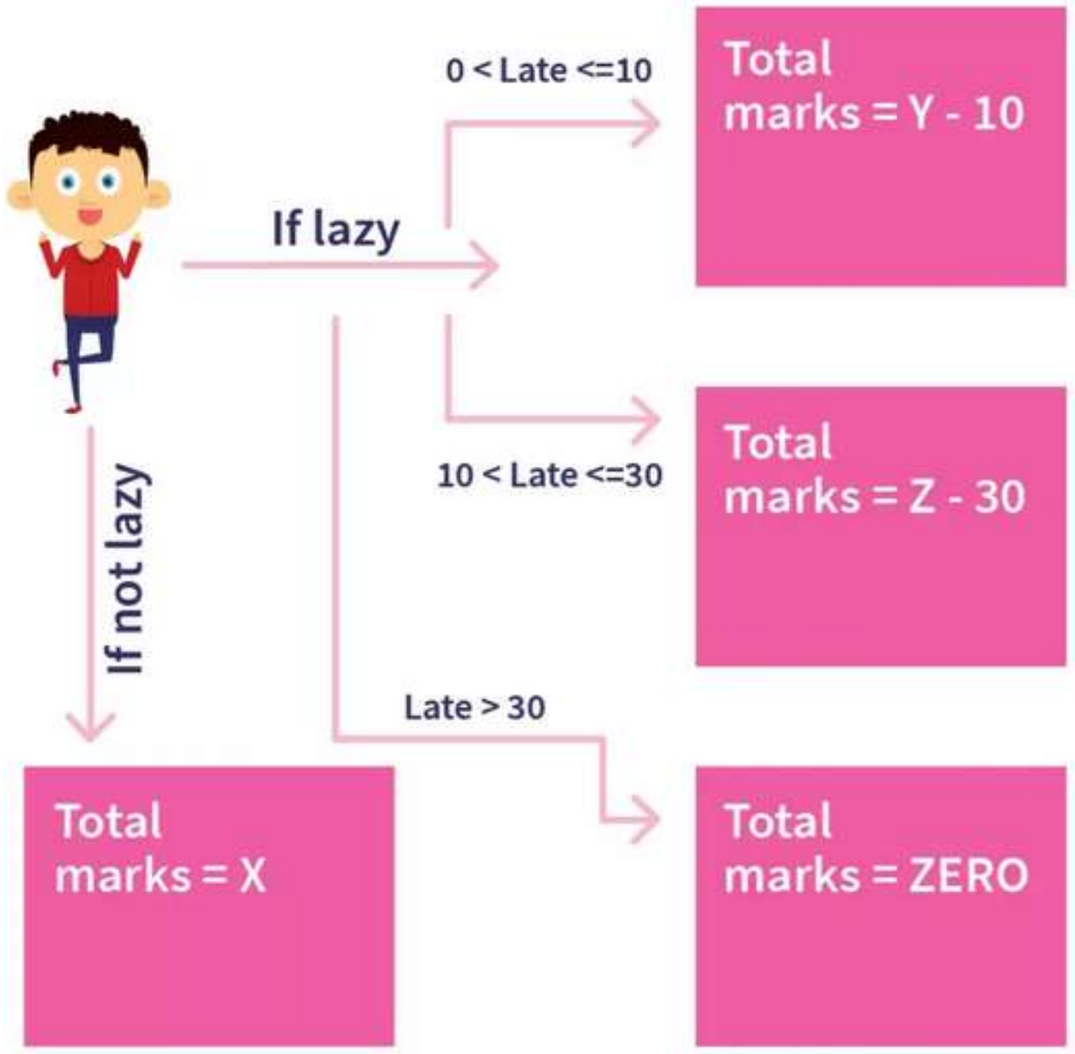
steps to achieve it (in the ideal case of course). Similarly, in the coding world also, we make decisions and according to the situation, we follow required statements and skip irrelevant ones.

Let’s understand the flow control in python through another example. Let’s imagine ‘n’ number of students who have to go to the examination hall to give their paper. But some of them are lazy and for lazy students, there are some special rules.

Special rules:

1. No marks deduction if arrived at the examination hall on time.
2. If 10 mins late, then 10 marks will be deducted.
3. If 30 mins late, then 30 marks will be deducted.
4. Zero marks if later than 30 mins.

Let’s understand this example with a diagram and we later discuss this same example with python code (once I introduce you to the type of flow control)



We depicted the same example through a diagram that says that if a student is not lazy, no marks will be deducted (X marks). If a student is 10 mins late then 10 marks will be deducted from the total marks gained (Y marks). Similarly, if it is later than 10 mins and appears under 30 minutes, 30 marks will be deducted from total marks (Z marks). Else, anyone who comes after 30 mins will be allocated zero marks.

Before implementing this example on python, let me just quickly discuss the importance of python flow control.

Types of Flow Control in Python

Conditional statements and Loops

Conditional

if | if…. else | if…. elif…. else

Let’s understand them one by one using the exam problem defined above.

For simplicity, the default time is 9 units. 10 minutes late would be 19 units and 30 minutes would be 39 units.

if Condition

```
time = 19
if time == 9 :
   print("On time")
if time > 9 and time <= 19:
   print("10 minutes late")
if time > 19 and time <= 39:
  print("30 minutes late")
if time > 39:
  print("Zero marks")
```

See the code above. We wrote the code using multiple if conditions. It runs in a synchronous manner. This means it'll check if condition i.e. time == 9 and then second if condition and so on. If the condition becomes true then it'll print the subsequent print statement else it'll ignore it and move to the next if condition (or end of the program)

time = 19,

1. Since time is not equal to 9, therefore, first if condition becomes false.
2. Given time lies under 2nd if condition. So, it'll become true and it'll print "10 minutes late"
3. 3rd if condition will be false because given time doesn't lie under this range.
4. Similarly, 4th if condition will also become false.

So, only the second if condition will become true and it'll print "10 minutes late".

This kind of writing if condition is useful in many areas but it's not necessary here because in this way it'll check each and every if condition irrespective of whether the prior has already become true or not. In this example also, it's checking each if condition even when the first if condition becomes true and that's redundant. Isn't it?

To solve this, we have other ways of writing the same code.

if else Condition

```
time = 21

if time == 9 :
   print("On time")
else:
    if time > 9 and time <= 19:
        print("10 minutes late")
    else:
        if time > 19 and time <= 39:
            print("30 minutes late")
        else:
            if time > 39:
                print("Zero marks")
```

Let's take time = 19 and see how this optimizes the above code example.

1. First if condition will become false because given time 19 is not equal to 9
2. Moving to else,

2.1 First if condition will become true because given time 19 lies under the given range time > 9 and time <= 19. Therefore, it'll run and print "10 minutes late".

2.2 Since, first if condition under else block becomes true, therefore, further else block and conditional statements related to it will not execute. Whereas in just if conditional statements, it's checking each if condition even if one if condition becomes true.

The difference between this code and the above is that we're skipping the entire block of code if it's not necessary to check. Let's see another example.

time = 21,

So, first the condition will become false because time is not equal to 9.

Then it'll move to another block and check first if the condition in this block i.e. time > 9 and time <= 19. This is also false because time is equal to 21 (given) hence, it's false. Then it moves to the next else block and checks the first if-condition present in it. Now, this if condition will become true because time is equal to 21 and the condition is if time > 19 and time <= 39 then run this block. So, it'll print "30 minutes late". But, it'll not check the other if condition i.e. if time > 39.

This is a very useful technique to write conditional statements but according to the problem we're solving this is still not the best way to write it. You can also see how we made the simple-looking code more complex using this unnecessarily. Though it's quite an improvement (performance-wise) but our code is not clean (which is also necessary).

Let's improve it with the next conditional statement.

if elif else Condition

```
time = 40

if time == 9 :
   print("On time")

elif time > 9 and time <= 19:
   print("10 minutes late")

elif time > 19 and time <= 39:
  print("30 minutes late")

else:
  print("Zero marks")
```

Looks clean but does it work? Let's see.

time = 40

Clearly, first if condition will become false. But imagine what will happen if time is equal to 9? If time is equal to 9, the first if-condition will become true and it'll print "on time". Now, it'll simply reach the end of the code and never ever reach other conditional statements defined here. Do you see the improvement now? Back to the original given time i.e. 40.

When time is equal to 40 and the first if-condition becomes false, it'll move to the next elif statement (else if) i.e. time > 9 and time <= 19. Clearly, it's also false. Similarly, other elif statements will also become false. Once, all elif statements become false and we are left with nothing except an else block then all the statements defined inside the else block will run. In this example, it'll print "Zero marks".

Let's take one more example.

time = 29,

1. First if condition will become false since time is not equal to 9
2. First elif condition will also become false because time doesn't lie under a given range.
3. Second elif condition will become true because time i.e. 29 lies under given range – time > 19 and time <= 39
4. Since, code found a true conditional statement, therefore, further conditional statements will not be executed. In the above example, the last else condition will not be executed because of the same reason

So, we're getting the different outputs according to which conditional statement is true. We discussed 3 types of conditional statements but one way is more appropriate than others according to the problem we're solving.Till now, we're working with an example of one student. In the problem statement, it's defined that we've to perform this same task (which we

did above) for ‘n’ number of students.Say, if n = 10, one way to implement it is that we repeat our code ten times and add an additional if condition to monitor the current value of the student.

```
students_arrival_time = [9,25,39,45,9,75,84,2,18,13]
student_counter = 0
if student_counter == 0:
    if students_arrival_time[student_counter] == 9 :
        print("On time")
        student_counter+=1
    # all conditional statements here…
if student_counter == 1:
    if students_arrival_time[student_counter] == 9 :
        print("On time")
        student_counter+=1

    # all conditional statements here...
if student_counter == 2:
    if students_arrival_time[student_counter] == 9 :
        print("On time")
        student_counter+=1
  # all conditional statements here
```

Before you get overwhelmed with this lengthy code, let me tell you it’s not the correct way to handle this. It’s written here just to make you understand why it’s wrong and how we can improve it.

We defined an array called students_arrival_time and a counter, student_counter to perform operations for each student. If you see closely, we make use of 1st and 3rd conditional statements i.e. if condition and if elif else. By this example, you can understand how we can combine them and how each one of the ways are important on their own.

We're doing the same thing as shown in the above code examples, the only change is that we used the 1st conditional statement (if condition) to check for which student we're performing the operation. Once a student is executed, we're incrementing the student_counter and because we're using 1st conditional statement (if condition), it'll not stop once a condition becomes true, it'll check other if conditions also. So, using it we're specifying other if conditions to handle operation for the next student and so on.

BUT as you can see, it's not the optimal way to handle it. I showed you the code for 2 students and you can assume how lengthy it would become if I'd done it for 10 students. In reality, we've not just 10 but more than 1K students in just a single campus. So, do you think it's the right way? Also, in programming, we have got a DRY principle. Don't Repeat Yourself principle which says that you shouldn't write the same piece of code multiple times in multiple places if it's doing the same thing. And, there are multiple ways to follow this. Obviously, discussing all the ways is out of the scope of this article but there is one way which we can discuss. Yeah, that's LOOPS!

Loops (For Loop | While Loop)

Generally, a loop consists of three things. Initialization, condition, and incrementation.

Initialization is used to initialize the starting point of the loop from where it starts performing operations. It could be index, counter, or maybe you could define a range like [2, n] so, that loop will start from 2. And, according to the problem you can modify the loop, there is no compulsion over how you define your initializer as long as it's syntactically correct.

Condition, it's the same as we learned above. It is used to define the ending point of the loop. Leaving it empty will make the loop run infinitely until the memory limit of the system exceeds this because the loop doesn't have any condition specified to come out of the loop.

Incrementation is used to increment the variable (index, counter, etc) to some steps. Leaving it empty can also make the loop run infinite because if we don't increment the variable, it'll keep executing operations for the same variable.

Infinite Loop Problem

Let's understand this with the problem we're trying to solve using loops (the 'n' students' problem).

total_students = 10 (assumption)

If we set **student_counter** to 1 and don't increment it then it'll keep running for the same student every time (never ending loop). If we increment it by 1 every time but don't specify

the condition to stop the loop when student_counter reaches 11 (because we have only 10 students) then also it'll run infinitely. Got it now?

Let's implement this problem using a for loop.

For Loop

```
students_arrival_time = [9,25,39,45,9,75,84,2,18,13]

for student_counter in range(len(students_arrival_time)):
    if students_arrival_time[student_counter] == 9 :
        print("On time")
    elif students_arrival_time[student_counter] > 9 and
students_arrival_time[student_counter] <= 19:
        print("10 minutes late")
    elif students_arrival_time[student_counter] > 19 and
students_arrival_time[student_counter] <= 39:
        print("30 minutes late")
    else:
        print("Zero marks")
```

All the code part is same except the for loop statement. Let's understand this statement in reverse order.

```
for student_counter in range(len(students_arrival_time)):
```

len() gives the length of the array. In our example, it's used on the students_arrival_time array which has a length of 10.

range(n) function provides first n integers starting from 0 till n-1. It'll auto increment it by 1 step. In our example, student_counter will become 0 then 1 till the length of array i.e 10 so, student_counter will have value from 0 to 9.

Now, we have defined the variable, student_counter. Initialized it with 0. Loop will run until student_counter doesn't exceed the length of the array (condition). Using **range(),** will auto-increment it by 1 step. So, that's how we're performing the same operation on 10 different students with the help of just 1 line.

A for loop is also used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages. With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

```
courses = ["PHP", "C", "JAVA", "Python", "JS"]
for course in courses:
    print(course)
    if course == "Python":
        break
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
courses = ["PHP", "C", "JAVA", "Python", "JS"]
for course in courses:
    if course == "JAVA":
        continue
    print(course)
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(10):
    print(x)
```

Note that **range(10)** is not the values of 0 to 6, but the values 0 to 9.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(5, 10), which means values from 5 to 10 (but not including 10):

```
for x in range(5, 10):
    print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 20, **2**):

```
for x in range(2, 20, 2):
    print(x)
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

```
for x in range(1,6):
    print(x)
else:
    print("Finally finished!")
```

Note: The **else** block will NOT be executed if the loop is stopped by a break statement.

```python
for x in range(7):
    if x == 5: break
    print(x)
else:
    print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```python
for x in range(1,4):
    for y in range(1, 11):
        print(x, 'X', y, '=', x*y)
    else:
        print('End of multiplication for', x)
    print('--------------------------')
```

The pass Statement

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
for x in [0, 1, 2]:
    pass
```

While Loop

```
students_arrival_time = [9,25,39,45,9,75,84,2,18,13]

student_counter = 0
while student_counter < len(students_arrival_time):
    if students_arrival_time[student_counter] == 9 :
        print("On time")
        student_counter+=1
    elif students_arrival_time[student_counter] > 9 and
students_arrival_time[student_counter] <= 19:
        print("10 minutes late")
        student_counter+=1
    elif students_arrival_time[student_counter] > 19 and
students_arrival_time[student_counter] <= 39:
        print("30 minutes late")
        student_counter+=1
    else:
        print("Zero marks")
```

We initialized the **student_counter** to zero first and then define the condition. We're not incrementing the counter in the same line and this is where the difference between both lies.

Generally, all the code written using for loop can be done using while loop but it has some special use cases.

While loop gives more control to the programmer than for loop (though this is debatable and varied according to the programmer's style of writing the code). This is because we have the liberty to increment or decrement the counter according to the code requirement. Whereas in for loop, we have to pre-define the increment or decrement counter steps.

With the while loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 10:
    print(i)
    i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement

With the break statement we can stop the loop even if the while condition is true:

```
i = 1
while i < 10:
    print(i)
    if i == 5:
        break
    i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

```
i = 0
while i < 10:
  i += 1
  if i == 5:
    continue
  print(i)
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

```
x = 1
while x < 10:
  print(x)
  x += 1
else:
  print("x is no longer less than 10")
```