



UNILAG

INST0002 PROGRAMMING 1 & INST0091 INTRODUCTION TO PROGRAMMING

LECTURE 03 — CONTROL FLOW STATEMENTS

Dr Daniel Onah
Lecturer
Module Leader

Copyright Note:

Copyright Licence of this lecture resources is with the Module Lecturer named in the front page of the slides. If this lecture draws upon work by third parties (e.g. Case Study publishers) such third parties also hold copyright. It must not be copied, reproduced, transferred, distributed, leased, licensed or shared with any other individual(s) and/or organisations, including web-based organisations, online platforms without permission of the copyright holder(s) at any point in time.

Recap

- We discussed about variables and constants
- We looked at primitive data types
- We discussed good code layout and comments
- Arithmetic expressions and assignment statements
- We wrote programs to read and process inputs
- We discussed about Strings in Python

Re: Cap☺



Outline

- Conditional statement: IF, ELIF, ELSE
- For Loop Iteration
- While Loop
- Nested statement
- Operators and indentation in Python
- Summary

Conditional statement: IF Statement

Equality and Relational Operators

It is important for us to revisit these operators again as they are essential for our program construct using conditional statements.

Condition:

- An expression that can be **true** or **false**.
- If selection statement
 - Allows a program to make a **decision** based on a condition's value
- Equality operators (**==** and **!=**)
- Relational operators (**>**, **<**, **>=**, and **<=**)
- Both equality operators have the same **level of precedence**, which is **lower** than that of the **relational operators**.
- The **equality operators** associate from **left to right**.
- The **relational operators** all have the same *level of precedence* and also associate from **left to right**.

Equality and Relational Operators

Algebraic operator	Python equality or relational operator	Sample Python condition	Meaning of Python condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

The IF Statement

The **IF statement** allows us to write our program into **logical sequences of code** that run only if **certain condition has been met**. This makes the program to be robust by allowing only a portion of code to run if a certain condition is true or has been met.

Most conditional statement such as the **IF** construct comes under the umbrella of decision making. This simply means the program are written using a certain **decision driven statement**, therefore executing a particular set , or sets of code based on given conditions.

In principle, an if statement lets your program know whether or not, it should execute a block of code.

The IF Statement

In Python, the IF statement is a basic decision-making statement. It is used to decide whether a certain statement or combination of statements will be executed or not within the code block, only if a certain condition is true, then a **single statement , or block of statements** is executed, otherwise (or ELSE) it will not be executed.

The syntax for the conditional IF statement is:

```
if(condition):
```

```
    # The statement is to be executed if condition is true
```

The IF Statement

It is important to remember to add the colon (`:`) after the If statement. The colon denote that the immediate statement is inside the **if-statement-block** and create the indentation block for the code.

Example:

```
if(condition):  
    # if-statement-block  
    # statement to execute goes here
```

If you missed the colon, this would lead to a **SyntaxError**. This is also applicable to all conditional control statements including **while loop**, **for loop** and so on.

Example Program

Let's write a **single selection IF statement** program to check whether a number is smaller than another.

```
# variable number is initialized to 25.  
    number = 25  
# if-condition is checked, using  $25 < 50$ , the outcome result is true  
    if(number < 50):  
# “25 is less than 50” gets printed out.  
        print(“25 is less than 50”)  
# Note that if statements executes and considers one statement by default
```

Example Program

Let's take a look at another example.

```
number = 19

# if block statement to check whether the number is equal to 19
if(number == 19):
    number += 1 # then increment number by 1
    print("number = {}".format(number))
```

Output: number = 20

Prime number program

A prime number is a whole number that is divisible by itself. That is a number that can only be divided by itself and 1 without remainders. This is number greater than 1. The factors of a prime number are 1 and itself.

Test:

If given a set of numbers or randomly generated numbers, how would you print out the prime numbers from the set.

Exercise 1

Write a program to find whether an integer is an even number. If the number is an even number, then print: "The given number is an even number" and stop the execution. If the integer is not an even number, then print nothing.

Even number program: Solution

```
number = 18
# check if number is even
if(number % 2 == 0):
    print("The given number is an even number")
```

IF-THEN Statement

The IF-Then Statement

The **if-then** and **if-then-else** conditional statements are extensions of the **if statement** and as such, allows Python to make simple decisions based upon the environment.

Scenario: Let's see a real-life thought process and decision making. Say you want to make a dinner arrangements to go out with your friend and you say to yourself: “If my friend gets home before 19:00PM then we will go out for the dinner”. When 19:00PM arrives, the condition you set, i.e., friend is at home, will determine whether you go out for the diner or not.

In other words, it will be **true** or **false**. It is this thought process that works in the exact same way in Python.

The IF-Then Statement

Case Study: A ticket master at an ice hockey club want to drive ticket sales and wishes to advertise that anyone attending the hockey match next week gets 20% if they are under 20 years of age, when they booked online. The program need to compute whether the customer purchasing the ticket is eligible for the 20% discount on the ticket price.

We can let the program make the decision for us using the if-then statement below:

```
if(age < 20):  
    isCustomer = True
```

Case Study: Description

- In our program, an integer variable called **age** was used to store the age of the customer for the ticket.
- The condition for the program (i.e., is the ticket purchaser **under 20**) is placed inside an open and closed brackets.
- Now, if this condition is true, the the statement beneath the if statement block is executed. In this case, we set the **BOOLEAN** variable of the **isCustomer** to **TRUE** and this is an important part, as the condition must equal a **BOOLEAN** value.

Case Study: Description

You would notice that quite often a Python program needs to **execute more than one statement at a time if the condition is true**. We can achieve this by writing the statement immediately at the next line within the **if-statement** block as shown below.

```
age = 17
if(age < 20):
    isCustomer = True
    discount = 20
    print("You are eligible for a discount of {}".format(discount))
```

The IF-THEN-ELSE Statement

An if-then statement can be extended to have statements that are executed when the condition is **false**, and the example below illustrates this.

```
if(condition):  
    #execute statement(s) if condition is true  
else:  
    #execute statement(s) if condition is false
```

Example: Hockey ticket (complexity)

In the previous example about the ice Hockey ticket program. Let's add a **slight complexity** to the code. Now, **we need to ensure that everyone else (customers) with age 20 and above should get a zero discount**. Let's look at how we could add this additional complexity to the code.

Note, the condition is that this program will execute the **else block** if the customer's age is **20** and **above**.

```
age = 20
if(age < 20):
    isCustomer = True
    discount = 20
    print("You are eligible for a discount of {}".format(discount))
else:
    discount = 0
    print("You are not eligible for a discount therefore you got {}".format(discount))
```

The IF-THEN-ELSE Statement

The **if-the-else** statement also allows the **nesting** of the **if-then statement** and we will be discussing nesting later on, but for now, the example below demonstrates this:

```
# nested if-then-else

if(age < 20):
    isCustomer = True
    discount = 20
elif(age > 67):
    isPensioner = True
    discount = 10
elif(isStudent == True):
    discount = 5
```

The IF-THEN-ELSE Statement

As you can see in the previous program, the **if-then-else** statement pattern just **repeats itself**. If at any time the condition is **true**, then the relevant **statements are executed** and any other **conditions beneath are not tested** to see whether they are even **true or false**. The program just terminate at the point with the correct execution or with the correct option that met the program **conditional requirements**.

Let's look at the **isStudent == true** condition. This condition is written so that we can test and verify whether or not the **isStudent** has a value of true or not, as been declared using a Boolean variable. For example:

```
elif(isStudent):  
    discount = 5
```


Exercise 2

Write a Python program to print **'Pass'** if a student's score is ≥ 50 or **'Fail'** if the marks scored is < 50 . Use **If-Elif**.

Exercise 2 - Solution

Write a Python program to print **'Pass'** if a student's score is ≥ 50 or **'Fail'** if the marks scored is < 50 . Use If-Elif.

```
marksScored = 79
grade = " "
if(marksScored >= 50):
    grade = "Pass"
elif(marksScored < 50):
    grade = "Fail"
print("Grade is = ", grade)
```

Exercise 3

Let's assume students have been given a coursework to work on. Write a program to check if they have completed the task. If they have completed the given task display **“Very good, Well done!”**. But, if they are still working on the given task display **“Good work, continue!”**.

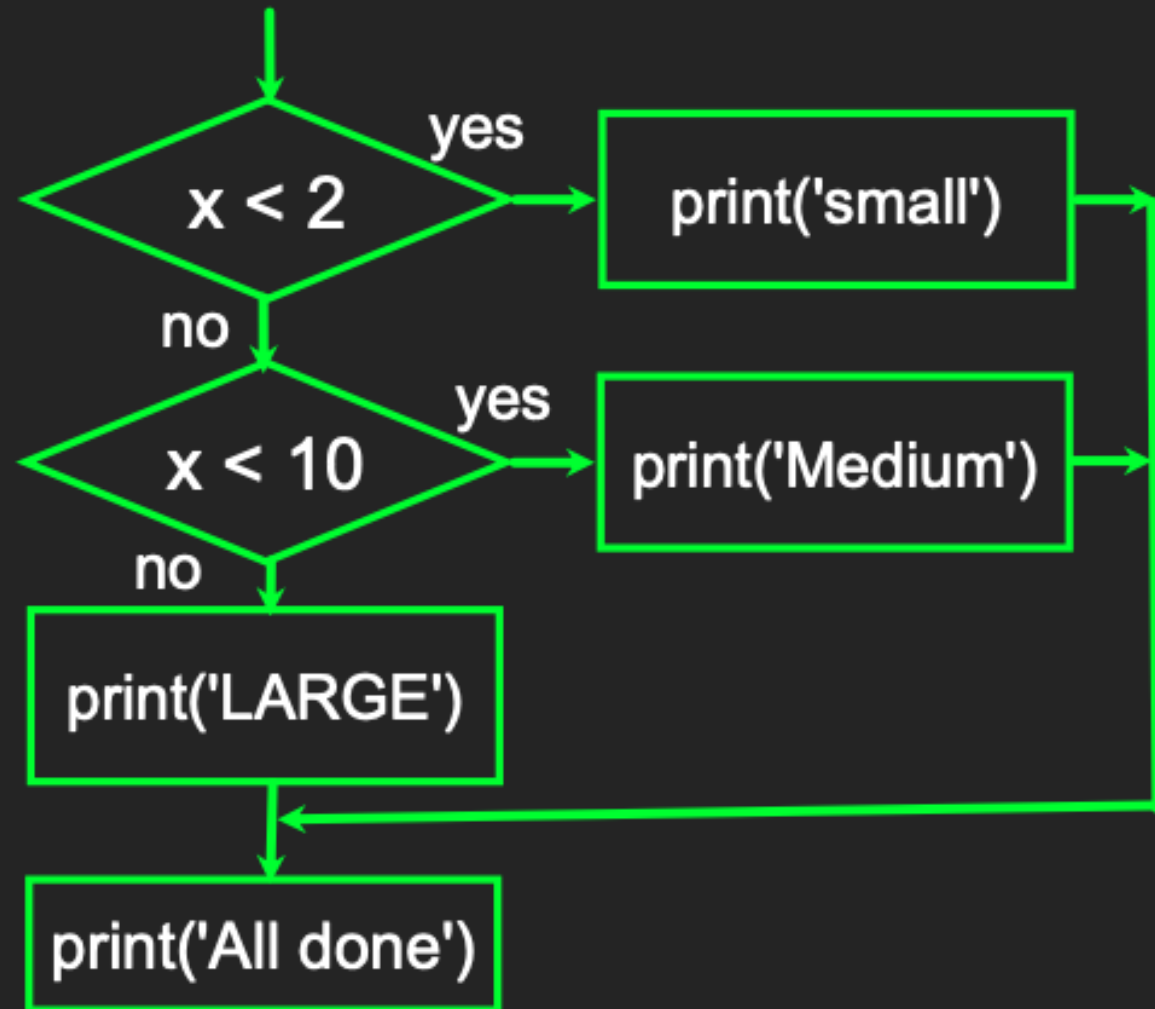
Exercise 3 - Solution

Let's assume students have been given a coursework to work on. Write a program to check if they have completed the task. If they have completed the given task display **“Very good, Well done!”**. But, if they are still working on the given task display **“Good work, continue!”**.

```
taskCompleted = 'y'
if(taskCompleted == 'y'):
    print("Very good, Well done!")
elif(taskCompleted != 'y'):
    print("Good work, Continue")
```

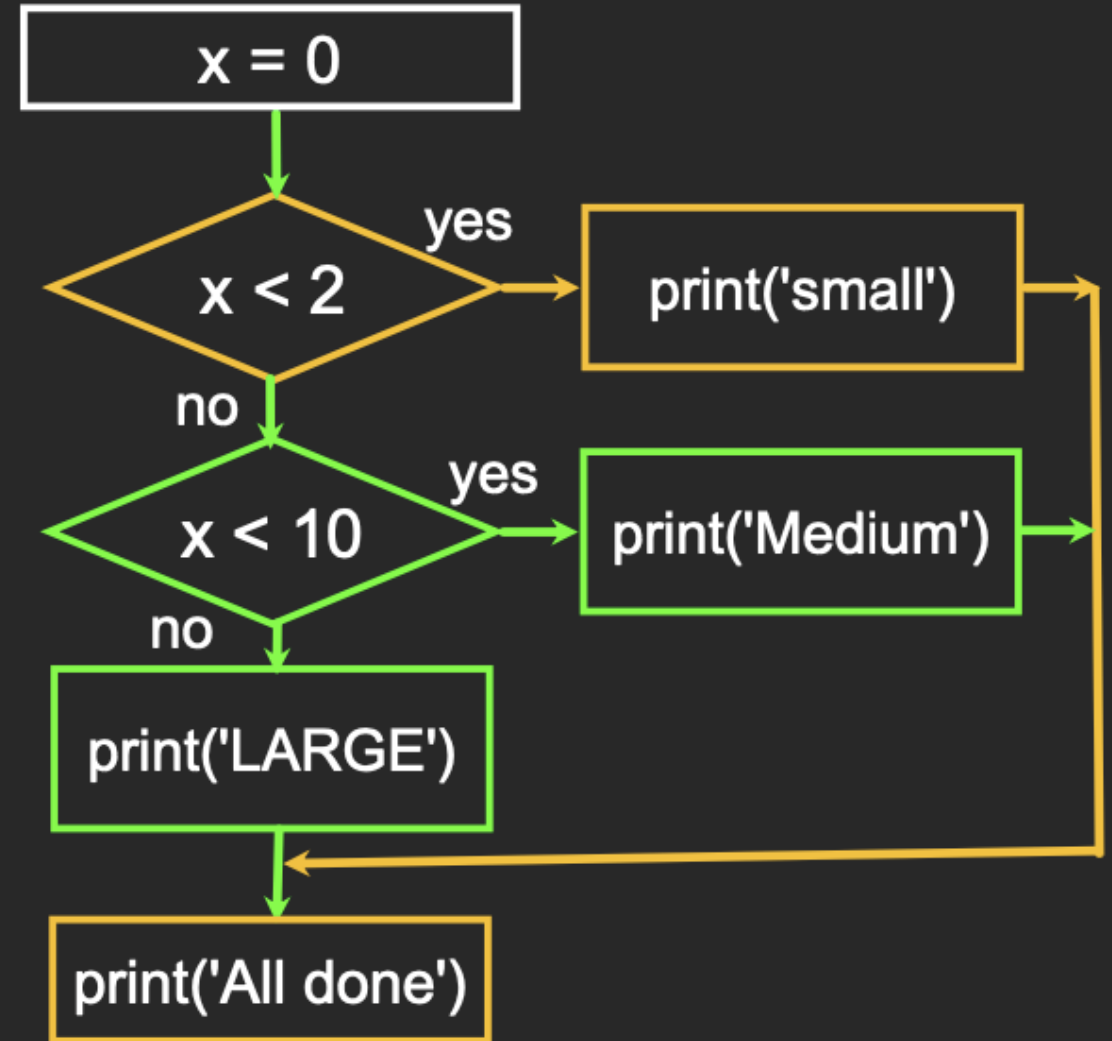
Multi-way

```
if x < 2 :  
    print('small')  
elif x < 10 :  
    print('Medium')  
else :  
    print('LARGE')  
print('All done')
```



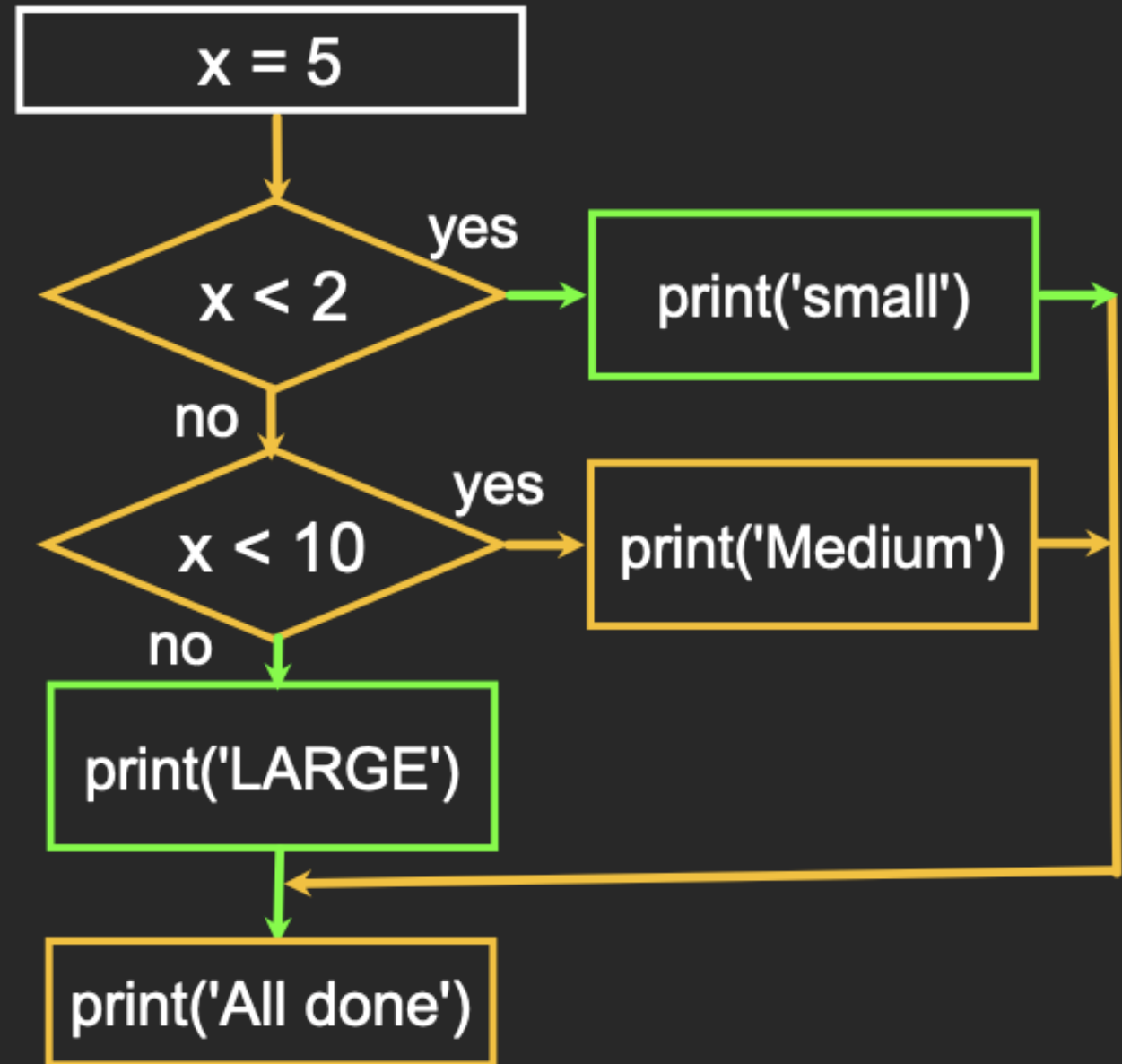
Multi-way

```
x = 0
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



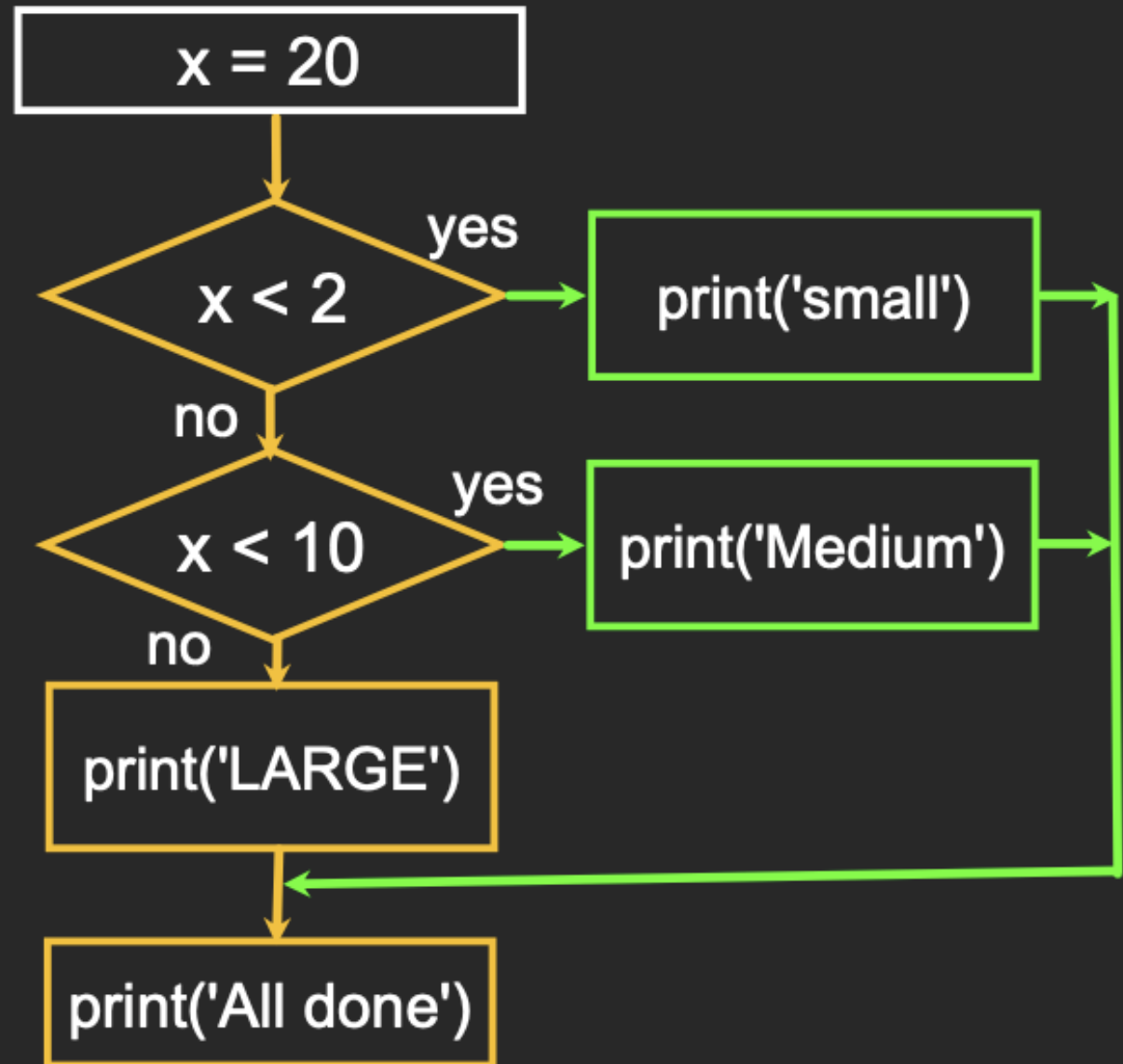
Multi-way

```
x = 5
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Multi-way

```
x = 20
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Multi-way

```
if x < 2 :  
    print('Small')  
elif x < 10 :  
    print('Medium')  
elif x < 20 :  
    print('Big')  
elif x < 40 :  
    print('Large')  
elif x < 100:  
    print('Huge')  
else :  
    print('Very Massive')
```

```
# No Else  
x = 5  
if x < 2 :  
    print('Small')  
elif x < 10 :  
    print('Medium')  
  
print ('All done')
```

Multi-way Puzzles: Home Exercise

An exercise for you to try. Check if this block of codes will print regardless of the value of x.

```
if x < 2 :  
    print('Below 2')  
elif x >= 2 :  
    print('Two or more')  
else :  
    print('Something else')
```

```
if x < 2 :  
    print('Below 2')  
elif x < 20 :  
    print('Below 20')  
elif x < 10 :  
    print('Below 10')  
else :  
    print('Something else')
```

IF-ELSE statement

The IF-ELSE Statement

The **if-else conditional statement** allows a much greater control over the flow of the program code than **basic if statement**, which was detailed in the first part of this lecture, and it does this by allowing multiple tests to be grouped.

An **else** clause in a code will be executed if the condition in the **if** statement fails or result to a **false**.

The great thing you should note about the **else statement** is that it can **proceed another if test**, which is very useful when multiple, mutually exclusive tests need to be run at the same time. Each test will move onto the next and then to the next and so on until a **true** test is encountered.

When a true test is found, its associated block of code is executed.

The IF-ELSE Statement

In the **if else test**, once the associated code is run, the program will then skip to the **following or next-line of the if-else statement**. If none of the tests **proves to be true**, the default else block is executed, if there is one defined. This mean the program has set to the default behaviour if every other conditions fail.

Generic syntax:

```
if (condition1):  
    # do first thing  
elif (condition2):  
    # do second thing  
else:  
    # do third thing
```

The IF-ELSE Statement

Let's illustrate this in another way.

Generic syntax:

```
if (condition):  
    # Executes this block if condition is true  
else:  
    # Executes this block if condition is false
```

The IF-ELSE Statement

Now that we have discussed the advantages of the else statement and presented a few generic examples, let us run through a few additional examples to help us visualize this concepts a little better.

Generic syntax:

```
# x is initialized to 49
x = 49
if (x < 36): # if-condition is checked. 49 < 36 yields false
    print("x is smaller than 49")
# flow enters the else block
else:
    print( "x is greater than 36")
```

The IF-ELSE Statement

Now that we have discussed the [advantages of the else statement](#) and presented a few generic examples, let us run through a few additional examples to help us visualize this concepts a little better.

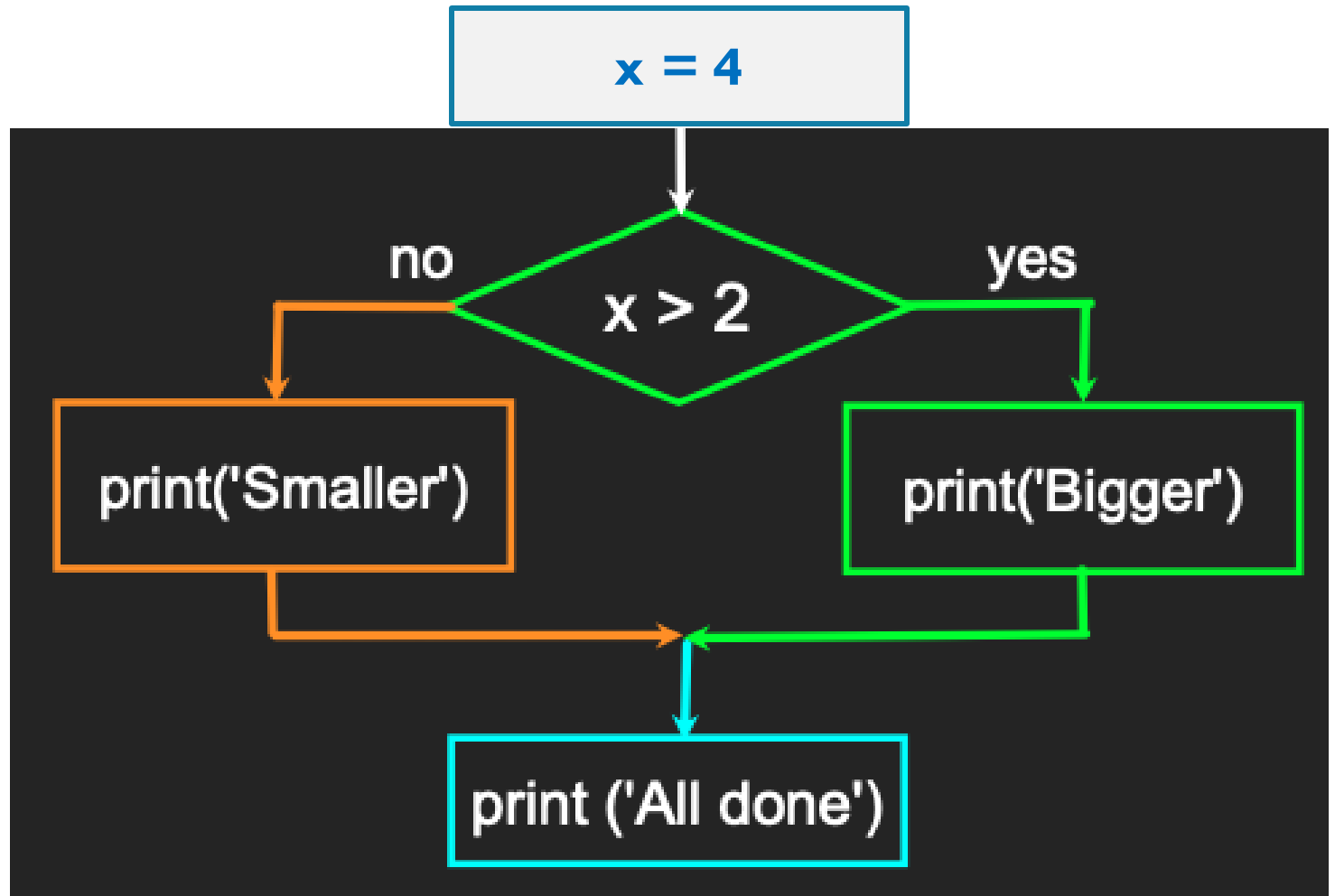
Generic syntax:

```
# university is initialized as a string
university = "University College London"
if (university == "Hello UCL"): # if-condition is checked. Is university equal to Hello UCL
    print(" Hello UCL")
# flow enters the else block
else:
    print( "Welcome to University College London")
```


Two-way Decisions

Sometimes we want to do one thing if a logical expression is **true** and do something else if the expression is **false**

Here, we need to choose one thing or the other, but not **both two-way at the same time**.

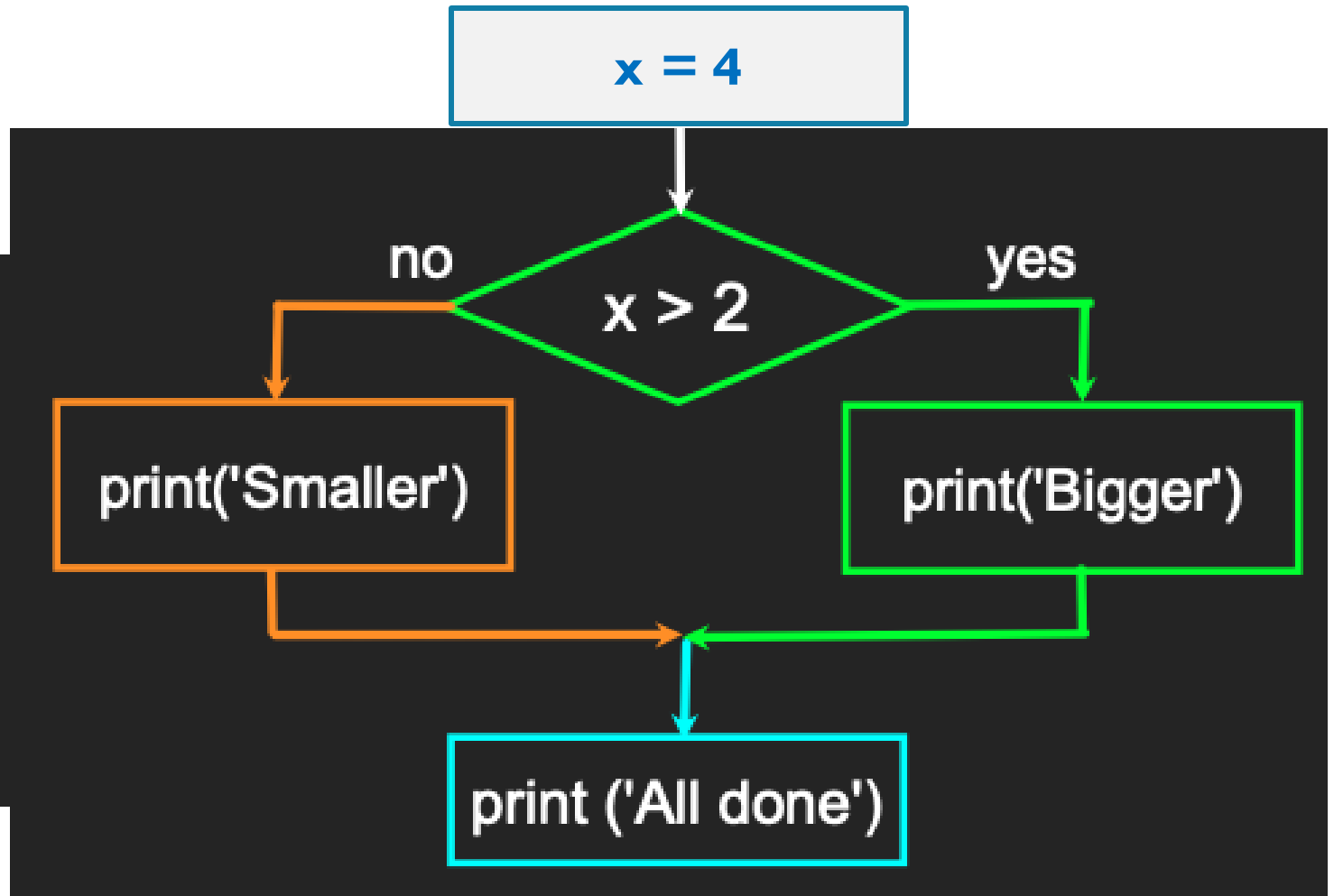


Two-way Decisions Using **Else**:

```
x = 4

if x > 2 :
    print('Bigger')
else :
    print('Smaller')

print ('All done')
```

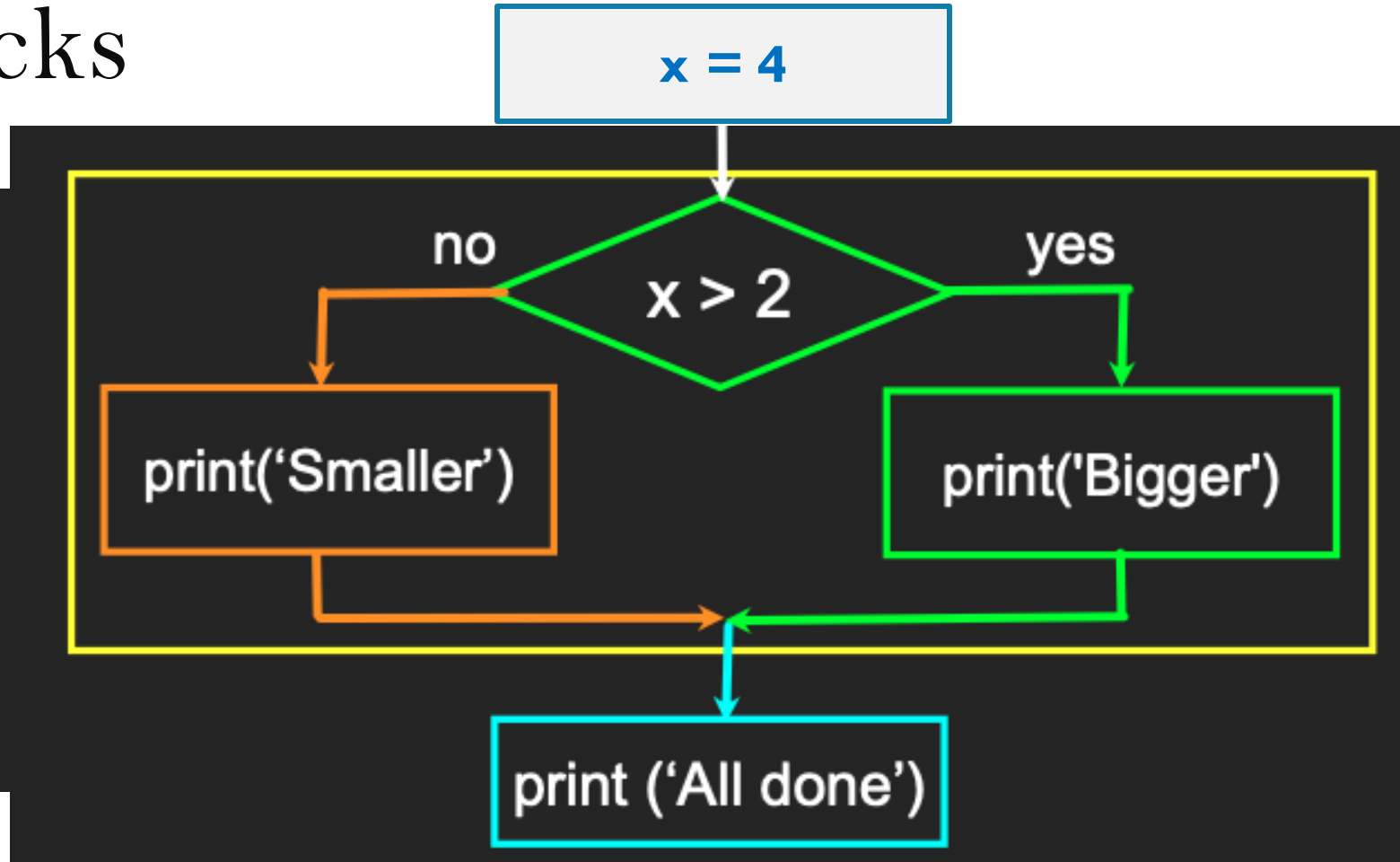


Visualizing Blocks

`x = 4`

```
if x > 2 :  
    print('Bigger')  
else :  
    print('Smaller')
```

```
print ('All done')
```



Nested Statement

The NESTED Statement

What are nested statements and why do we need them in Python programming?

A nested statement is a statement that we **write upon another conditional statement**. We define a statement and within the block of the statement, we define another statement. For a nested statement, only when the first statement is true, before the next statement within its block will execute.

The Logic Behind NESTED Statement

The logic behind the nested statement is that the program code first evaluates the **outer statement** and if this (**outer**) statement is **false**, then the inner statement **will not be executed**.

However, if the **outer** statement is **true**, then the inner statement will be executed and **print out** the result of the statement.

In other words, the **nested statement**, evaluates the outer conditional statement and **only when it succeeds and met the conditions** before it could successfully evaluate the **inner conditional statement**.

NESTED-IF Statement

Let's discuss what this **nested-if** statement simply means. A nested if is an **if statement at the outer condition** with another **inner if statement** within the block of the first **if** statement. This is simply another if statement **within the if-body block** or **within the else body block**.

For example:

```
# outer if
if( number > 13): # this first condition must be true then
    # inner if
    if(number < 29): # before the second condition is executed and
        print("number is between 13 and 29") # print the result
```

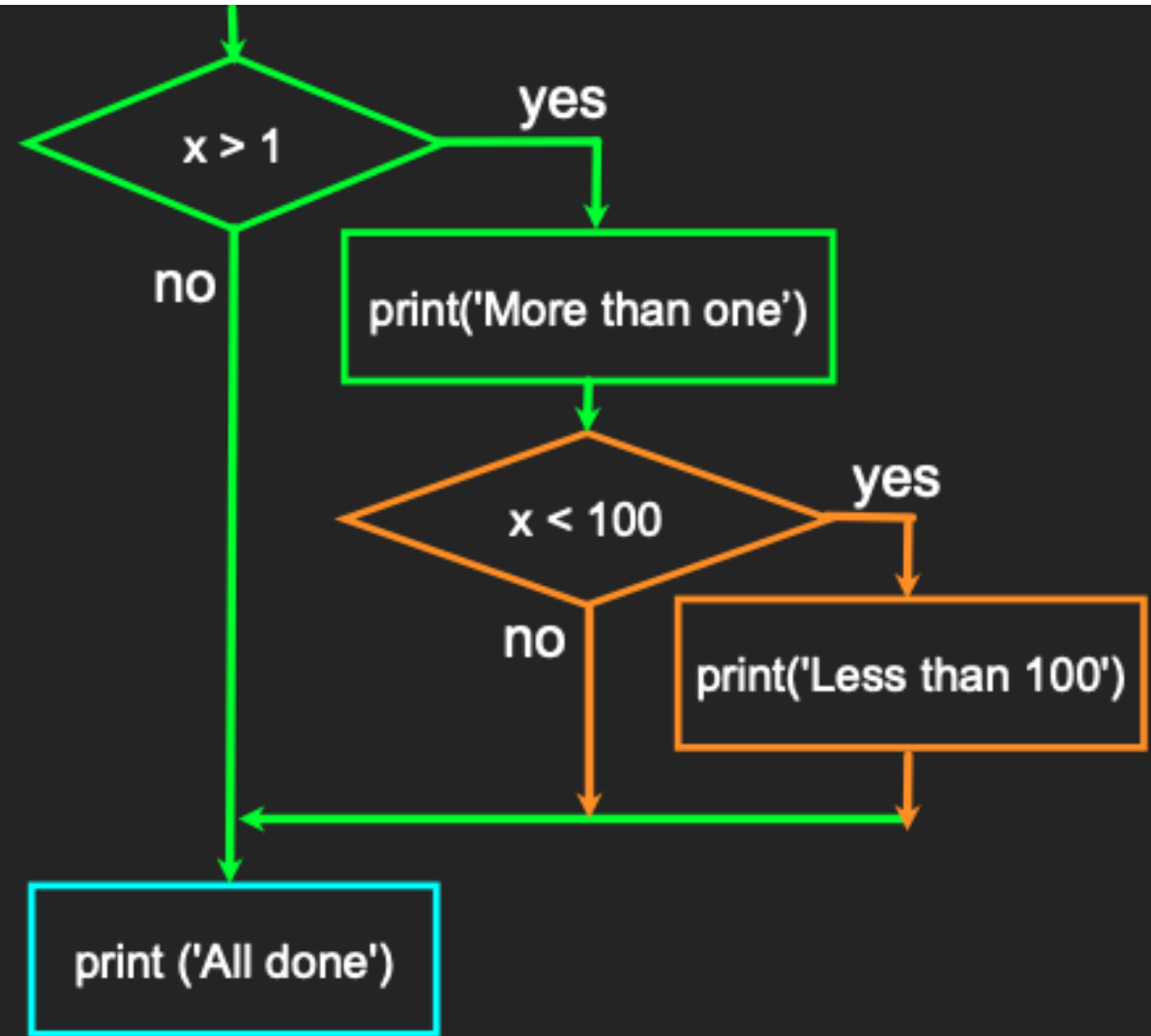
NESTED-IF Statement

In case, where the **outer if-body contains more than one statements**, of which is in the inner **if-body**, then its very important to note that the **print statement** only depends on the **successful evaluation of the outer if statement**. This simply means only when the condition of the outer if statement is **true** before the print statement will print the result(s).

```
# outer if
if( number > 13): # only when this condition is true before any of the print statement will execute
    print("This is correct!")
    # inner if
    if(number < 29):
        print("number is between 13 and 29")
```


Nested Decisions

```
x = 42
if x > 1 :
    print('More than one')
    if x < 100 :
        print('Less than 100')
print('All done')
```



Example 1

Let's look at an example:

```
number = 57

# first outer if conditional statement
if(number < 70):
    print("number is less than 70")
    # the nested if statement below will only be executed if the statement above it is true
    if(number > 47):
        print("number is greater than 47")
else:
    print("number is greater than 70") # if the first condition is false, the else statement is executed
```

Output:

number is less than 70

number is greater than 47

Example 2

Note that if the first condition is **true** and the second condition is **false**, the program will only execute the print statement of the **first condition**. **Example:**

```
number = 37
# first outer if conditional statement
if(number < 70):
    print("number is less than 70")
    # the nested if statement below will only be executed if the statement above it is true
    if(number > 47):
        print("number is greater than 47")
else:
    print("number is greater than 70") # if the first condition is false, the else statement is executed
```

Output: number is less than 70

Example 3

Note that if the first condition is **false** and the second condition is **true** or **false**, then the program will only execute the print statement of the **else** condition. **Example:**

```
number = 90
# first outer if conditional statement
if(number < 70):
    print("number is less than 70")
    # the nested if statement below will only be executed if the statement above it is true
    if(number < 105):
        print("number is less than 105")
else:
    print("number is greater than 70") # if the first condition is false, the else statement is executed
```

Output: number is greater than 70

Nested within an Else Body

As a brief recollection, we have emphasized in the *nested if* statement that when an *if* statement is in the *if-body*, then both the *outer* and *inner if* conditions must be *true* before the inner *if-body* can run.

But what happens if an *if statement* appears in the *body* of the *else statement*.

```
# Nested within an else body
# outer if condition
if(number > 75):
    print("Congratulations you score an A")
else:
    if(number > 65): # the statement "Congratulation you score B" will only execute
#if the outer if condition is false and inner if condition within the else body is true
        print("Congratulations you score a B")
```

Nested within an Else Body

The statement “Congratulations you score a B” is only printed *if and only if* the outer if condition results in *false* and the inner *if* condition results in *true*. In other words, the first if condition must have to “fail” before the second *print statement* will print the outcome.

So if the first condition fails, it means that the number is not greater than 75. so it must be less than or equal to 75. The second condition denote that the number is greater than 65. To explain this better, if we try to combine the meaning of the two statements, *we have a number greater than 65, but less than or equal to 75.*

Note:

It is important to note that *Python goes through each conditional structure one at a time.* When the *interpreter reaches the first condition that results to true*, it runs or execute the **statement in that body** and then *ignores or skips everything (or codes) else after or afterwards.*

Example 4

In the program code below, you don't see "Congratulations you score a C" unless the **first condition fails** (i.e. `number > 75`), and the **middle condition fails** (i.e. `number > 65`), **but the last condition in the else statement block succeeds** (i.e. `number > 55`). To interpret this statement "Congratulations you score a C", we say the number should be less than or equal to *65* and greater than *55*. For example:

```
# nested within an else body
number = 60
if(number > 75): # this condition fails
    print("Congratulations you score an A") # not printed
elif(number > 65): # this condition fails
    print("Congratulations you score a B") # not printed
else:
    if(number > 55): # this condition succeeds
        print("Congratulations you got a C") # before this will be printed
```

Exercise 4

Write a Python program to evaluate whether a number is **less than zero**, or **greater than zero** or **equal to zero**.

```
number = 33
if(number < 0): # check if number is less than zero
    print("The number is less than zero")
elif(number > 0): # check if number is greater than zero
    print("The number is greater than zero")
else:
    #if(number == 0): # this would also work successfully
    print("The number is zero")# check if the number is zero
```

Output: The number is greater than zero

Logical **and/or**

In Python programming we represent logical notation of **and** and **or** to indicate certain conditions. One of which must be **true** (**and operator**) before the program will run and the other either condition is **true or false** (**or operator**) before the program will execute.

Logical operator	Meaning
and	This simply means both conditions must be true before the program will execute.
or	This simply means if either of the condition is true or false , the program should execute.

Exercise 5

Write a Python program to find if the value of a given number is in units (numbers from: 1 - 9), **tens** (numbers from: 10 - 99), **hundreds** (numbers from: 100 - 999) and **more or otherwise** (numbers from: 1000 and above).

```
number = 299
if(number < 10 and number >= 1):
    print("The value of the number is in units")
elif(number < 100 and number >= 10 ):
    print("The value of the number is in tens")
elif(number < 1000 and number >= 100):
    print("The value of the number is in hundreds")
else:
    print("The value of the number is greater than hundreds")
```

Output: The value of the number is in hundreds

Facts

We have seen that it is possible to **nest** one *if-else* statement inside another *if-else* statement. It gives you the ability to specify conditions and to continue the execution expanding upon the result obtained by checking those **conditions set within the code block**. Nested *if/if-else* statements are used extensively throughout good conditional code structures.

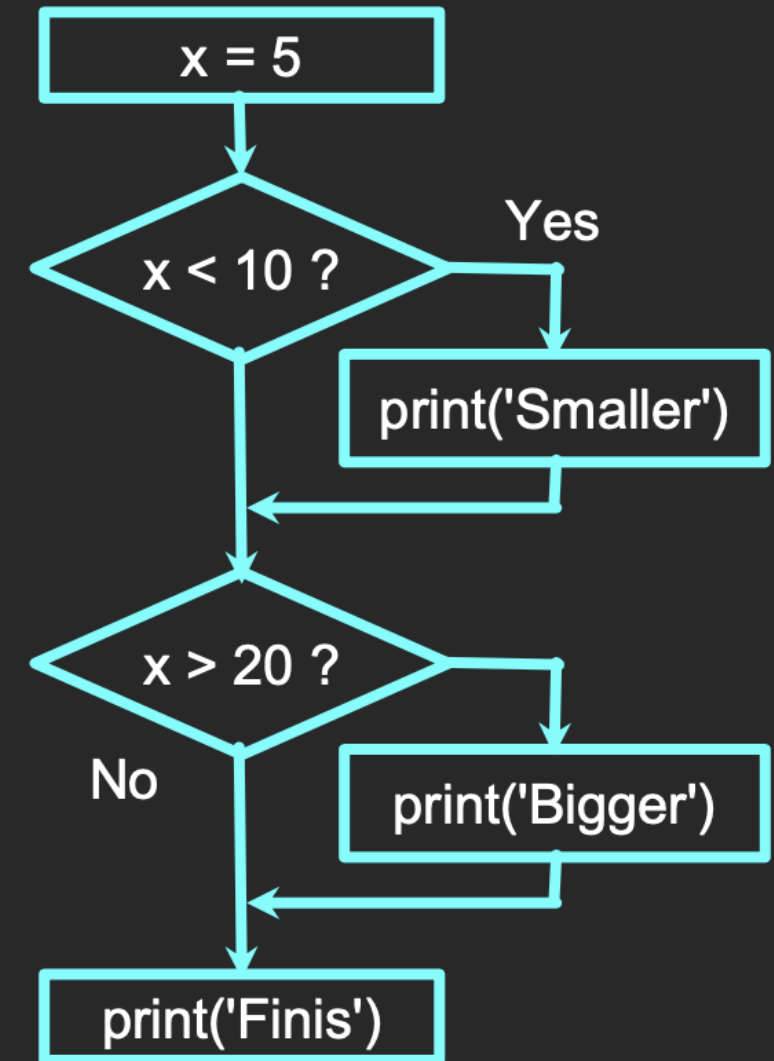
Conditional Flows

Program:

```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')
print('Finis')
```

Output:

Smaller
Finis



For Loop

Introduction to Loops

The **Python loops** are used to *iterate* part of a program several times and there are *two types of loops* in Python to be covered here: *The For Loop and While Loop*. Loops are programming element that *repeats a portion of the code a **set number of times***, they *iterate* until the desired true process is *complete*. They are essential in any program construct to *save time and minimize possible error*. Loops allow us to shorten hundreds of lines of codes to just a few lines, therefore making our program code far more efficient.

We will be discussing each loop individually with examples.

Iteration

Before we plunge right into the discussion about loops. It is essential for us to know about key terms such as **iteration**.

By definition, the term iteration means the process of instructing the Python program to repeat the code construct or steps over and over again.

Iteration therefore is synonymous with **LOOPS**, and this is an important process to include within our program codes. Why is this necessary in the structure of program such as codes for an algorithm. Well algorithms are inherently made up of *instructions that are performed one after another* and often, algorithm need to repeat certain steps until it is instructed to stop, or a particular application condition is met.

Iteration Steps in real-life

Let's look at a real-life scenario of preparing and eating a cornflakes cereal.

- 1) *pour cornflakes cereal into a bowl*
- 2) *add milk to the bowl of cereal*
- 3) *spoon milk and cereal into your mouth*
- 4) *repeat the step 3 until the bowl of cornflakes cereal is empty*
- 5) *end*

For Loop

We will begin loops exploration with the ***FOR LOOP control structure***. The For Loop is primarily used for a fixed iteration of a certain part of a Python program. In other words, you know how many times you want your program to repeat or iterate through a particular block of code. If we want to repeat some instructions according to a particular condition, this ***for loop*** control structure or *looping* will be used.

Generic syntax of For Loop:

```
for statement (condition):  
    # code block to be executed
```

For Loop – Examples-Range()

Let's look at a few examples of a for loop using range function.

```
for n in range(5): # print numbers from 0 to 4
```

```
print(n)
```



Out put: 0 1 2 3 4

```
for n in range(3,6): # print numbers from 3 to 5
```

```
print(n)
```



Out put: 3 4 5

```
for n in range(3, 8, 2): # print numbers from 3 to 7, in step of 2
```

```
print(n)
```



Out put: 3 5 7

For Loop

If the condition set is `true`, then the loop will `iterate again`, if the condition results in false, then the `loop ends`.

For example:

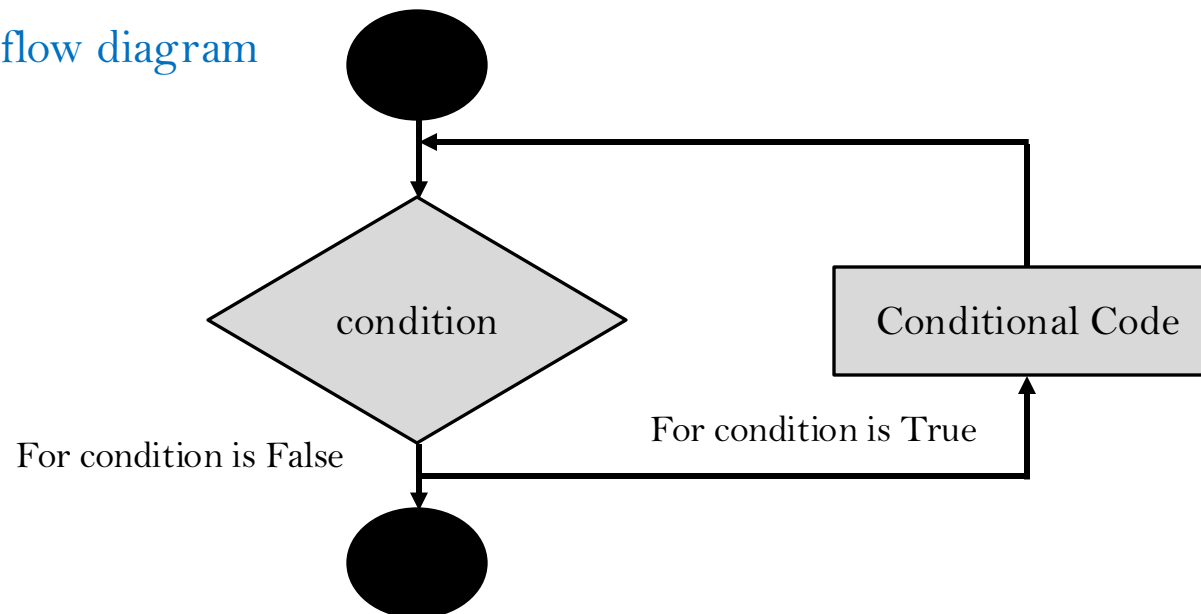
```
n = 10
for i in range(0, n):
    print(i)
```

Output: 0 1 2 3 4 5 6 7 8 9

For Loop

A *for loop* is used to repeat over a sequence that is either a *list*, *tuple*, *dictionary*, or a *set*. It is essential to note that the *in* keyword is part of the for statement's syntax and this functionally is unrelated to the *in* operator used as part of membership testing.

Example of a For-loop flow diagram



Quiz

Write a program using the 'for loop' to generate a list of values for :

$$y = 3x^2 + 3$$

where:

$$x = \{1, \dots, 5\}$$

Quiz - Sample Solution

Write a program using the 'for loop' to generate a list of values for :

$$y = 3x^2 + 3$$

where:

$$x = \{1, \dots, 5\}$$

```
for x in range(1, 5):  
    y = 3 * x ** 2 + 3  
    print(y)
```

For Loop

If the condition set is **true**, then the loop will **iterate again**, if the condition results in false, then the **loop ends**.

For example: **A list of numbers program.**

```
listOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for i in listOfNumbers:
    print(i)
```

Output: 1 2 3 4 5 6 7 8 9 10

For Loop

If the condition set is **true**, then the loop will **iterate again**, if the condition results in **false**, then the **loop ends**.

For example: print even numbers that are divisible by 2 and with a remainder of 0.

```
listOfNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

number = 2

for i in listOfNumbers:
    evenNumber = number * i # print even numbers
    print(evenNumber)
```

Output: 2 4 6 8 10 12 14 16 18 20

While Loop

While Loop

The While Loop differs slightly from *For Loop*. As a reminder, the *For Loop* is primarily used within **fixed iterations** of a certain part of our Python program. *While Loops* however, are primarily used where the iteration of a Python program are not fixed and will keep iterating until the block of code targeted results in **True condition**. It is this difference in feature that separates the *For Loop* and *While Loop*.

Note: Do not forget however to increase the iteration of the variables used. If not, the loop will continue on for ever! (*Infinite Loop*). To stop *infinite loop* use **Ctrl + C** on your keyboard.

Generic syntax for While Loop:

```
while(condition):  
    # code block to be executed goes here
```

While Loop

If we wish to write Python program to continuously request for input from user or to perform iteration until the condition is **TRUE**, then we can use the while construct below. If we wanted to infinitely iterate the **while Loop** we can also do that using the *while(True)* construct. However, we do need to execute the **Ctrl + C** command to terminate the loop.

Generic syntax for While Loop True:

```
while(True):  
    # code block to be executed goes here
```

Sentinel Controlled Loop

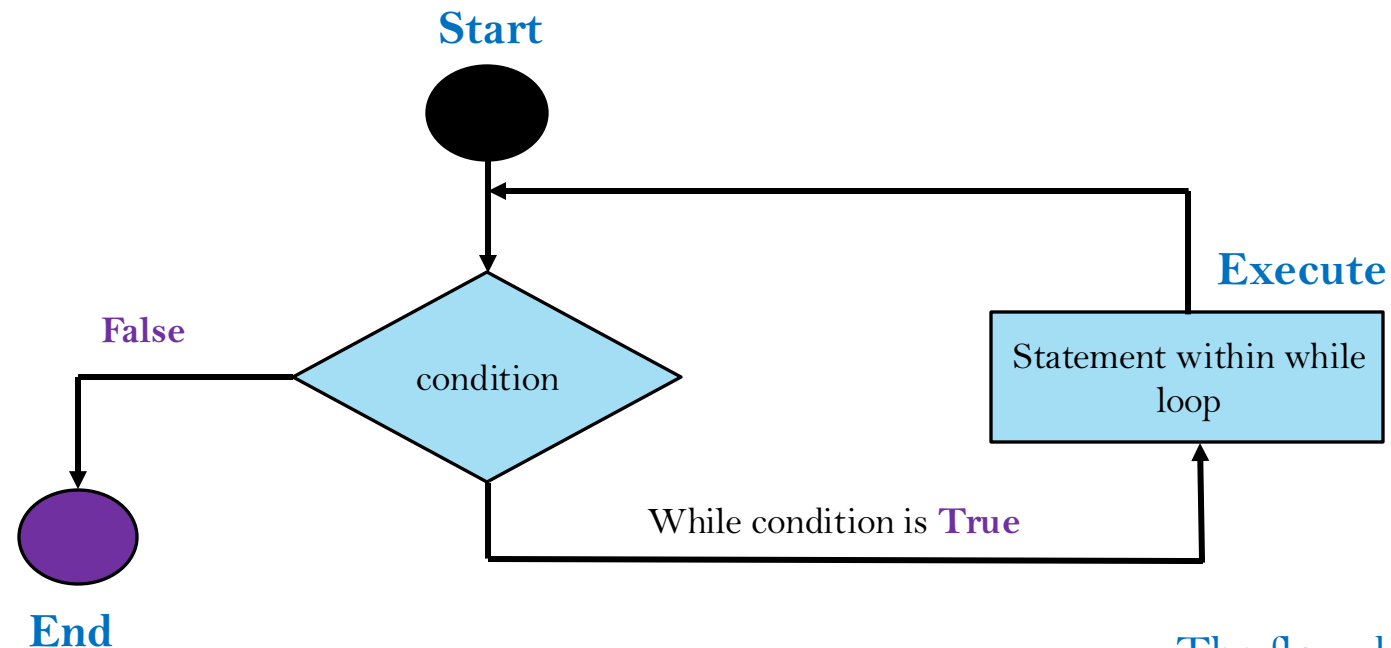
Sentinel controlled loop statement is a construct that we write into the while loop condition to instruct our program to stop looping once the sentinel value is entered as an input. The program will continue to execute just as in the while loop construct, but when we enter **this value (e.g., -1)** that we define within the while loop condition, then the program stops and print the outcome and so on. This *denotes the end of the loop process*, and this *sentinel-controlled* loop ***is not part of the data*** that is been generated by the while loop.

Generic syntax for **sentinel controlled** while loop:

```
while (number != -1):  
    # code block for execution goes here
```

While Loop

A **while loop** repeats the statement until a given condition is satisfied. If the condition is *true*, the *while loop control structure* will execute, otherwise, it won't.



The flow diagram of a while loop.

Quiz

Use a 'while' loop to generate a list of values of:

$$y = x^3$$

where:

$$x = -6 \text{ to } x = 6$$

Quiz – Sample Solution

Use a 'while' loop to generate a list of values of:

$$y = x^3$$

where:

$x = -6$ to $x = 6$

```
x = -6 # change this to zero and see the output
while(x <= 6):
    y = x ** 3
    print(y)
    x += 1 # increment x iteratively
```

While Loop: Example

Let's look at a typical while loop example.

```
counter = 0
while(counter < 10): # checking if condition is true
    print("The count is: ", counter)
    counter = counter + 1 # increment counter by 1
```



Output:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
The count is: 9
```


While Loop: Example

Let's look at a typical while loop example.

```
number = 2
while(number < 15): # checking if condition is true at
    each iteration
    print(number)
    number += 3 # increment number by 3
```

Output:



2
5
8
11
14

Nested Loop

A *nested loop* is a Python structure which simply means *one loop inside another loop*. We can place a *for loop inside a while loop* or a *while loop inside a for loop*, or *for loop inside a for loop*, or a *while loop inside a while loop*. Let's look at an example.

Task


Write a program that will use a nested for loop to find the prime number from 2 to 30 (exclusive).

Exercise & Solution

Write a program that will use a **nested for loop** to find the prime number from 2 to 30.

```
number = 2
while(number < 30): # checking against the condition
    num = 2
    while(num <= (number/num)):
        if not(number % num): # if there is no remainder
            break
        num = num + 1
    if(num > number/num):
        print(number, " is a prime number")
    number = number + 1 # increment the number by 1
```

Output:



```
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
23 is a prime number
29 is a prime number
```

Example

Let's look at a nested loop program to print a pattern.

```
for i in range(1, 5): # defining the range
    for j in range(i):
        print(j + 1, end = '')
    print()
```



Output:


```
1
1 2
1 2 3
1 2 3 4
```

Example

Let's look at a nested loop program to print a pattern.

```
char = '*'
suffix = "-->"
for i in range(1, 5): # defining the range
    for j in range(i):
        print(suffix, char, j + 1, end = "")
    print()
```

Output:



```
--> * 1
--> * 1 --> * 2
--> * 1 --> * 2 --> * 3
--> * 1 --> * 2 --> * 3 --> * 4
```

Practical Exercise

This exercise is based on Factorial computation. Factorial is a mathematical concepts that compute sequence of numbers by multiplying the previous number from the selected or inputted number to the last number which is 1. It is said to be used in power series for an exponential function. The factorial of a non-negative integers denoted by $n!$ is the product of all non-negative integers less than or equal to n .

Logically $n! = n \times (n-1) \times (n-2) \times (n-3)$

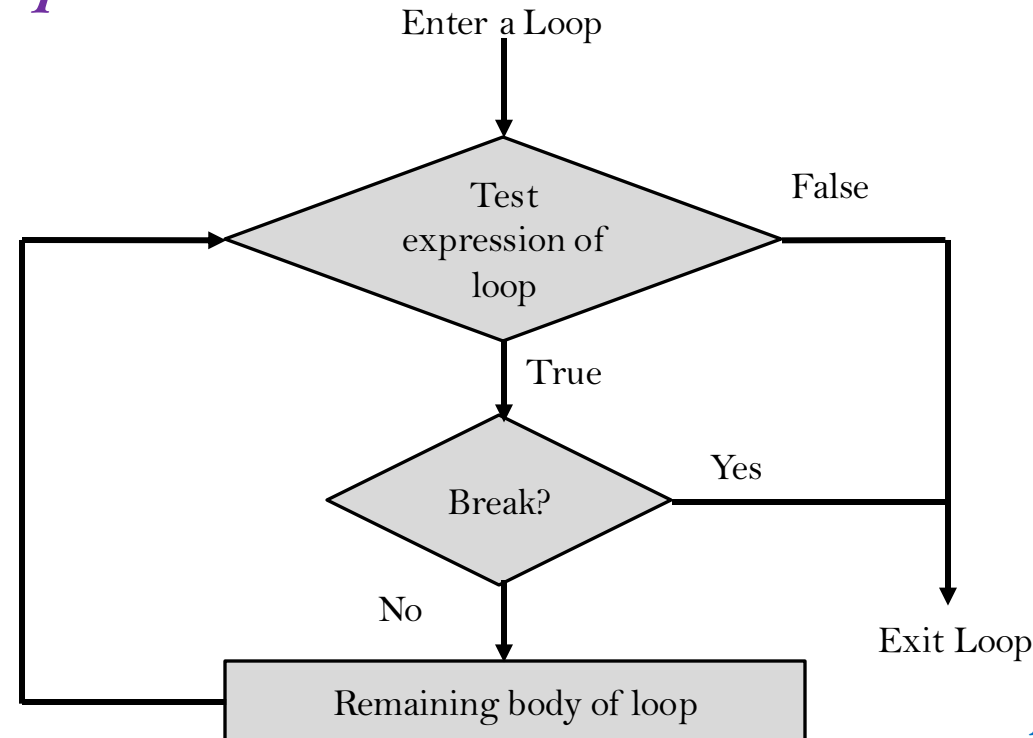
Example: 4 Factorial ($4!$) will be: $4 \times 3 \times 2 \times 1$ which is equal to 24.

Task:

Write a program that would find the factorial of eight ($8!$) using a while loop.

Break Statement

In Python, the break statement is used to *terminate a loop*. The control will switch out of the loop when the *break* statement is executed. If the break statement is inside a nested loop, then the break statement will *terminate the innermost loop*.



A flow diagram of a break statement

Example

A program to sum the first six numbers in a tuple.

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple
number_sum = 0 # initialising the variable 'number_sum' to zero
count = 0 # starting and initialising 'count' from zero
for x in numbers:
    number_sum = number_sum + x # add the number_sum value to first six numbers
                                together
    count = count + 1
    if count == 6: # counting and when the 6th number is reached stop
iteration/looping
        break # terminate the for loop
print("The sum of first ", count, "integers is: ", number_sum) # print the
                                                                outcome
```

Output: The sum of first 6 integers is: 21

Example

```
# example to break out of a loop if number 5 is reached
```

```
for i in range(10):
```

```
    print(i)
```

```
    if(i == 5):
```

```
        break
```

```
print(' Loop exited')
```

Output:

0

1

2

3

4

5

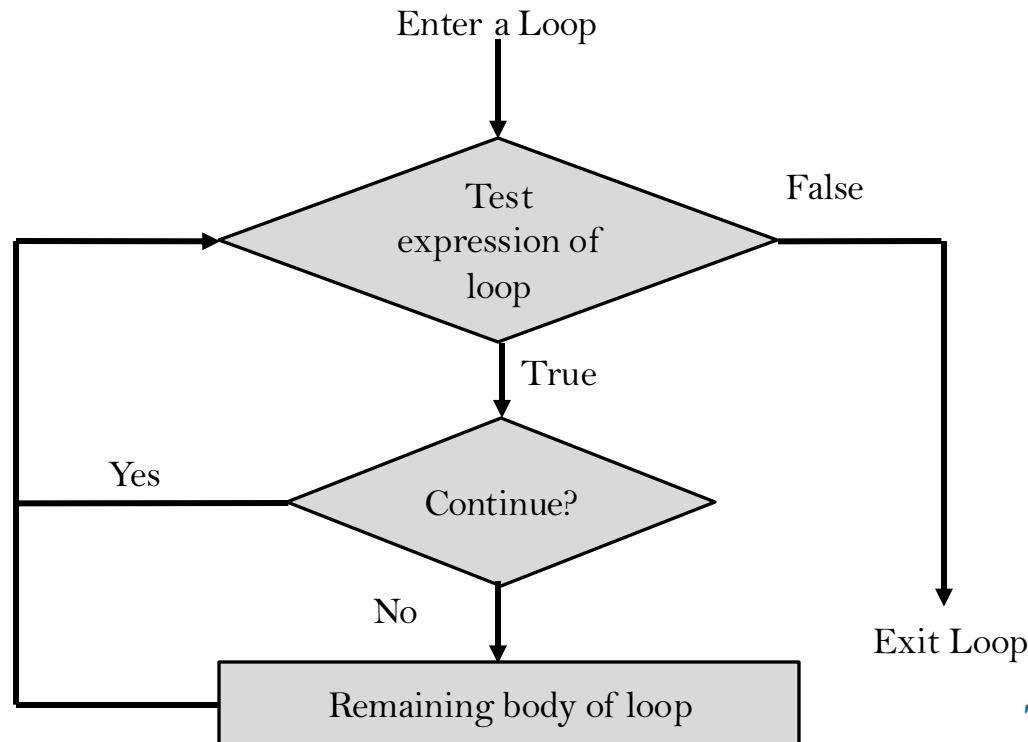
Loop exited

Continue Statement

In Python the continue statement is used to skip the rest of the code inside a loop for the current iteration only. In this continue statement, the loop does not terminate but continue with next iteration. In other words, the continue statement does not end the process within the loop. Unlike the break statement which terminates the loop process, the continue moves to the next available iteration to execute.

Continue Statement

The flow process of a continue statement in Python. It skips the condition if **true** and continue the execution of the rest condition.



The continue statement flow diagram

Example

```
# A program to skip a condition set e.g. if (i == 5).  
# The program will skip 5 (will not print 5) to print the rest of the numbers.  
  
for i in range(10):  
    if i == 5:  
        continue  
    print(i)
```

Output:



0
1
2
3
4
6
7
8
9

Example

```
# A program to skip a condition set e.g. if (i == 5 or x == 7).
```

```
# The program will skip 5 and 7 (will not print 5 or 7) to print the rest of the numbers.
```

```
for x in range(9):  
    if(x == 5 or x == 7):  
        continue  
    print(x)
```

Output:



0
1
2
3
4
6
8

Pass Statement

The `pass` statement in Python means a `null statement`. The major difference between a comment and a pass statement is that while the **interpreter ignores a comment entirely**, the **`pass` statement is not ignored**. Moreover, nothing really happens when the pass statement is executed. The result of a ***`pass` statement is a no operation*** (NOP). Let us take a look at a few examples to understand how the pass statement is used in Python.

```
number1 = 29
number2 = 56
if number2 > number1:
    pass
```

Output: No output (blank)

Pass Statement

The `pass` statement in Python means a `null statement`. The major difference between a comment and a pass statement is that while the **interpreter ignores a comment entirely**, the **`pass` statement is not ignored**. Moreover, nothing really happens when the pass statement is executed. The result of a ***`pass` statement is a no operation*** (NOP). Let us take a look at a few examples to understand how the pass statement is used in Python.

```
sequence = {'b', 'l', 'a', 'n', 'k'}  
for letter in sequence:  
    pass
```

Output: No output (blank)

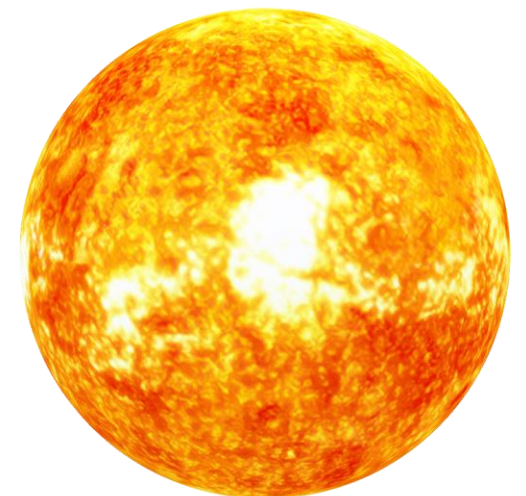
Comparison operators & Indentation

Comparison Operators

Boolean expression ask a question and produce a response of **Yes or No** result use to control program flow.

Boolean expression using comparison operators evaluate to **True/False** or **Yes/No**.

Comparison operators look at or access variables but **do not change** the values hold by variables or change the variables.



Comparison Operators

Python	Meaning
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal

Note: “=” is used for assignment operator

Comparison Operators

```
x = 5
if x == 5 :
    print('Equals 5')
if x > 4 :
    print('Greater than 4')
if x >= 5 :
    print('Greater than or Equals 5')
if x < 6 : print('Less than 6')
if x <= 5 :
    print('Less than or Equals 5')
if x != 6 :
    print('Not equal 6')
```

→ Equals 5

→ Greater than 4

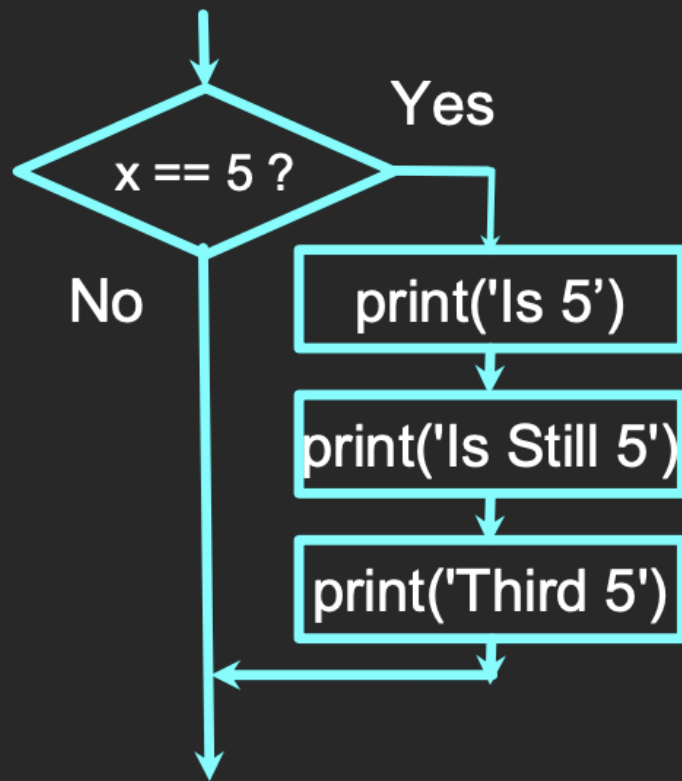
→ Greater than or Equals 5

→ Less than 6

→ Less than or Equals 5

→ Not equal 6

One-way Decisions



```
x = 5
print('Before 5')
if x == 5 :
    print('Is 5')
    print('Is Still 5')
    print('Third 5')
print('Afterwards 5')
print('Before 6')
if x == 6 :
    print('Is 6')
    print('Is Still 6')
    print('Third 6')
print('Afterwards 6')
```

Before 5
Is 5
Is Still 5
Third 5
Afterwards 5
Before 6
Afterwards 6

Indentation in Python

Python is a programming language that follows strict indentation approach to program constructs.

- **Increase indent** – indent after an **if** statement or **for** statement (after the colon “:”)
- **Maintain indent** to indicate the **scope** of the block of code (which lines are affected by the if/for statements).
- **Reduce indent** back to the level of the **if statement** or **for statement** to indicate the end of the block
- **Blank lines** are ignored – they do not affect **indentation**
- **Comments** on a line by themselves are ignored with regard to **indentation**

Warning: Turn Off Tabs!

Some editors and applications such as ‘**Atom**’ automatically uses spaces for files with “.py” extension (nice😊)


Most text editors can turn **tabs** into **spaces** – make sure to enable these features

- **NotePad++:** Settings -> Preferences -> Language Menu/Tab Settings
- **TextWrangler:** TextWrangler - > Preferences - > Editor Defaults

Python cares a *lot* about how far a line is indented. If you mix **tabs** and **spaces**, you may get “**indentation error**” even if everything looks fine

Structure: Indentation Code Block

increase / maintain after if or for
decrease to indicate end of block



The diagram shows a series of arrows on the left side of the code block, indicating the indentation level for each line. Yellow arrows point right for lines that increase or maintain the indentation level. Green arrows point right for lines that increase the indentation level. Orange arrows point left for lines that decrease the indentation level.

```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```

Colon (:) is used to start the indentation block in Python programs

Begin and End Blocks

```
x = 5
```

```
if x > 2 :
```

```
    print('Bigger than 2')
```

```
    print('Still gigger')
```


Begin and End Blocks

```
x = 5
```

```
if x > 2 :
```

```
    print('Bigger than 2')
```

```
    print('Still gigger')
```

```
print('Finish with 2')
```

Begin and End Blocks

```
x = 5
```

```
if x > 2 :
```

```
    print('Bigger than 2')
```

```
    print('Still gigger')
```

```
print('Finish with 2')
```

```
for i in range (5) :
```

```
    print(i)
```

Begin and End Blocks

```
x = 5
```

```
if x > 2 :  
    print('Bigger than 2')  
    print('Still gigger')
```

```
print('Finish with 2')
```

```
for i in range (5) :  
    print(i)  
    if i > 2 :  
        print('Bigger than number 2')
```

Begin and End Blocks

```
x = 5
```

```
if x > 2 :
```

```
    print('Bigger than 2')
```

```
    print('Still gigger')
```

```
print('Finish with 2')
```

```
for i in range (5) :
```

```
    print(i)
```

```
        if i > 2 :
```

```
            print('Bigger than number 2')
```

```
        print('Finish with i', i)
```

```
print('All Done')
```

Try and Except Block


Try/Except Block

You surround some part of your code with try and except

- If the code in the try works – the except is skipped
- If the code in the try fails – it jumps to the except block

Traceback Error

View program
code on console



```
$ cat errortry.py
greet = 'Hello Danny'
greetConv = int(greet)
print('First', greetConv)
greet = '123'
greetConv = int(greet)
print('Second', greetConv)
```

```
$ python errortry.py
Traceback (most recent call last):
  File "errorTry.py", line 2, in
    <module>
      greetConv = int(greet)
ValueError: invalid literal for int()
with base 10: 'Hello Danny'
```



Done

Traceback Error

The program
stops here

```
greet = 'Hello Danny'
greetConv = int(greet)
print('First', greetConv)
greet = '123'
greetConv = int(greet)
print('Second', greetConv)
```

Traceback (most recent call last):

**File "errorTry.py", line 2, in
<module>**

greetConv = int(greet)

**ValueError: invalid literal for int()
with base 10: 'Hello Danny'**

Done

Try/Except

```
$ cat tryexcept.py
```

```
greet = 'Hello Danny'
try:
    greetConv = int(greet)
except:
    greetConv = -1
print('First', greetConv)
greet = '123'
try:
    greetConv = int(greet)
except:
    greetConv = -1
print('Second', greetConv)
```

When the first conversion fails - it just drops into the except: clause and the program continues.

```
$ python tryexcept.py
First -1
Second 123
```

When the second conversion succeeds - it just skips the except: clause and the program continues.

Sample Try/Except Program

```
rawstr = input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work')
else:
    print('Not a number')
```

```
$ python3 trynum.py
Enter a number:42
Nice work
$ python3 trynum.py
Enter a number:forty-two
Not a number
$
```

Exercise 6

Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

Enter Hours: 45

Enter Rate: 10

Pay: 475.0

Note: Hours difference above 40 is $= 45 - 40$

Rate would be $= 10 * 1.5$

Then add them to the original expected rate & hours allocated to the employee.

$$475 = 40 * 10 + 5 * 15$$

Exercise 6 – Sample Solution

```
INCREMENT = 1.5
THRESHOLD_HOURS = 40
Hours = int(input("Enter Hours: "))
Rate = float(input("Enter Rate: "))
# write a condition to check if hours is more than 40
if (Hours > THRESHOLD_HOURS)
    differenceHours = Hours - THRESHOLD_HOURS
    #print(differenceHours) # uncomment to see the output
    rateIncrement = Rate * INCREMENT
    #print(rateIncrement) # uncomment to see the output
    payIncrease = (THRESHOLD_HOURS * Rate) + (differenceHours * rateIncrement)
    print("Pay = ", payIncrease)
else:
    pay = Hours * Rate
    print("Pay = ", pay)
```

Exercise 7

Rewrite your pay program using try and except so that your program handles non-numeric input gracefully.

Enter Hours: 20

Enter Rate: nine

Error, please enter numeric input

Enter Hours: forty

Error, please enter numeric input

Exercise 7 – Sample Solution

```
Hours = input("Enter Hours: ")
Rate = input("Enter Rate: ")
try:
    Hours = int(Hours)
    Rate = float(Rate)
    Pay = Hours * Rate
    print("Pay = ", Pay)
except:
    print("Error, please enter numeric input!")
    exit()
```

Exercise 7 – Sample Solution

```
try:
    Hours = input("Enter Hours: ")
    Hours = int(Hours)
except:
    print("Error, please enter numeric input!")
    exit()

try:
    Rate = input("Enter Rate: ")
    Rate = float(Rate)
except:
    print("Error, please enter numeric input!")
    quit()

try:
    Hours = int(Hours)
    Rate = float(Rate)
    Pay = Hours * Rate
    print("Pay = ", Pay)
except:
    print("Error, please enter numeric input!")
```

Summary

- We discussed control flow conditional statements: `if`, `if-elif`, `else`, `for` loop, `while` loop, `nested` loop
- We looked at comparison operators :- `==` , `<=` , `>=` , `>` , `<` , `!=`
- We discussed indentation
- Understanding One-way decisions
- We looked at Two-way decisions:- `if`: and `else`:
- Nested decisions for conditional statement & loops
- Multi-way decisions using `elif`
- `try/except` to compensate and handle errors

NOTE: MODULE GRADING SYSTEM

The rubric marks over 30 is not your final grade for the tutorial sheet. Each grade that you score is over 10. So e.g., if you score 22/30 for any of the tutorial sheets, you have to calculate the grade over 10 by subtracting 20. This will become 2/10. Note, any marks from 20 and below is automatic zero.

This is how your final grade will be calculated:

Grade Tutorial sheet 2 -> 6/10

Grade Tutorial sheet 3 -> 8 /10

Grade Tutorial sheet 4 -> 4/10

Grade Tutorial sheet 5 -> 8 /10

Grade Tutorial sheet 6 -> 8/10

Total grade = $6 + 8 + 4 + 8 + 8 = 34/50$

To make this percentage , multiply by 2. i.e. $34 * 2 = 68\%$

What to do Next

- Make sure you're familiar with how to enter, save and run a program.
- Check out the introductions for both the teaching concepts and coding strands.
- Check that you're familiar with where things are on the Moodle site and how to navigate around the resources.
- Practice-practice-practice! your Python programs using editors and command line.
- Start working on your tutorial sheet for the week.

Further Reading

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : By **Cay Horstmann & Rance Necaise**
 - Chapter 3 Decisions & Chapter 4 Loops
- Learning Python (5th Edition): By **Mark Lutz**
 - Chapter 12 If Test and Syntax Rules & Chapter 13 While and for loops
- Basic Core Python Programming: By **Meenu Kohli**
 - Chapter 8 Program Flow Control in Python

Next Lecture 4

Lecture 4 - Functions in Python