Flow Control

Program Control Flow

A program's control flow is the order in which the program's code executes.

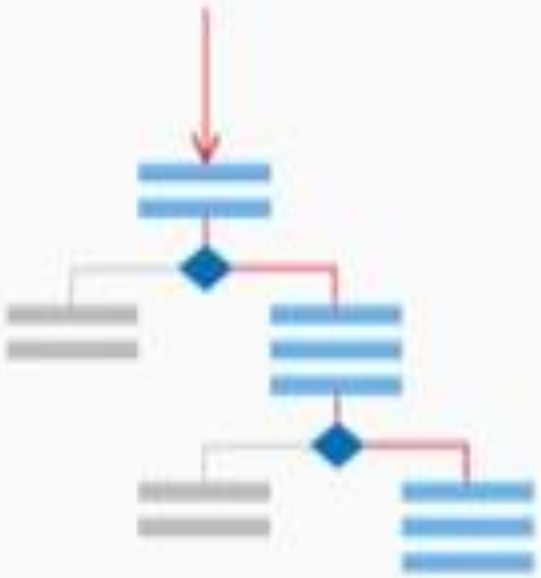
The control flow of a Python program is regulated by conditional statements, loops, and function calls. Raising and handling exceptions also affects control flow; exceptions will be covered in a later chapter.

Python has three types of control structures:

- Sequential - default mode
- Selection - used for decisions and branching
- Repetition - used for looping, i.e., repeating a piece of code multiple times.

Sequential

Control flows through all the statements, in the order in which it is written



Selection

Based on some conditions, control flows to different set of statements

Iteration

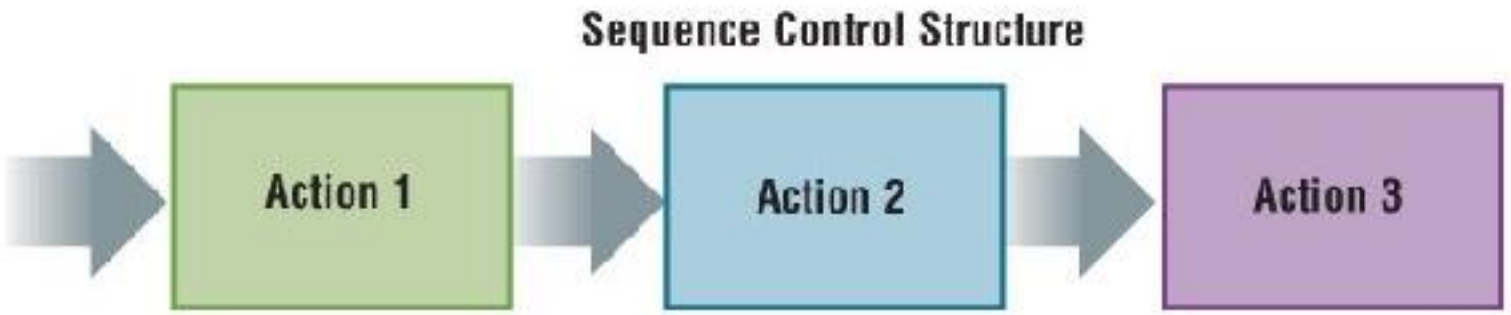
Certain statements will be executed repeatedly

Sequential

“Sequence control structure” refers to the line-by-line execution by which statements are executed sequentially, in the same order in which they appear in the program.

They might, for example, carry out a series of read or write operations, arithmetic operations, or assignments to variables.

Sequential Control Structure




```
a = 2;
b = 3;
c = a * b;
```

Selection/Decision control statements

In Python, the selection statements are also known as Decision control statements or branching statements.

The selection statements allows a program to test several conditions and execute instructions based on the conditon.

Some Decision Control Statements are:

- Simple if
- if-else
- if-elif-else (if_elif)
- nested if

Repetition

A repetition statement is used to repeat a group(block) of programming instructions.

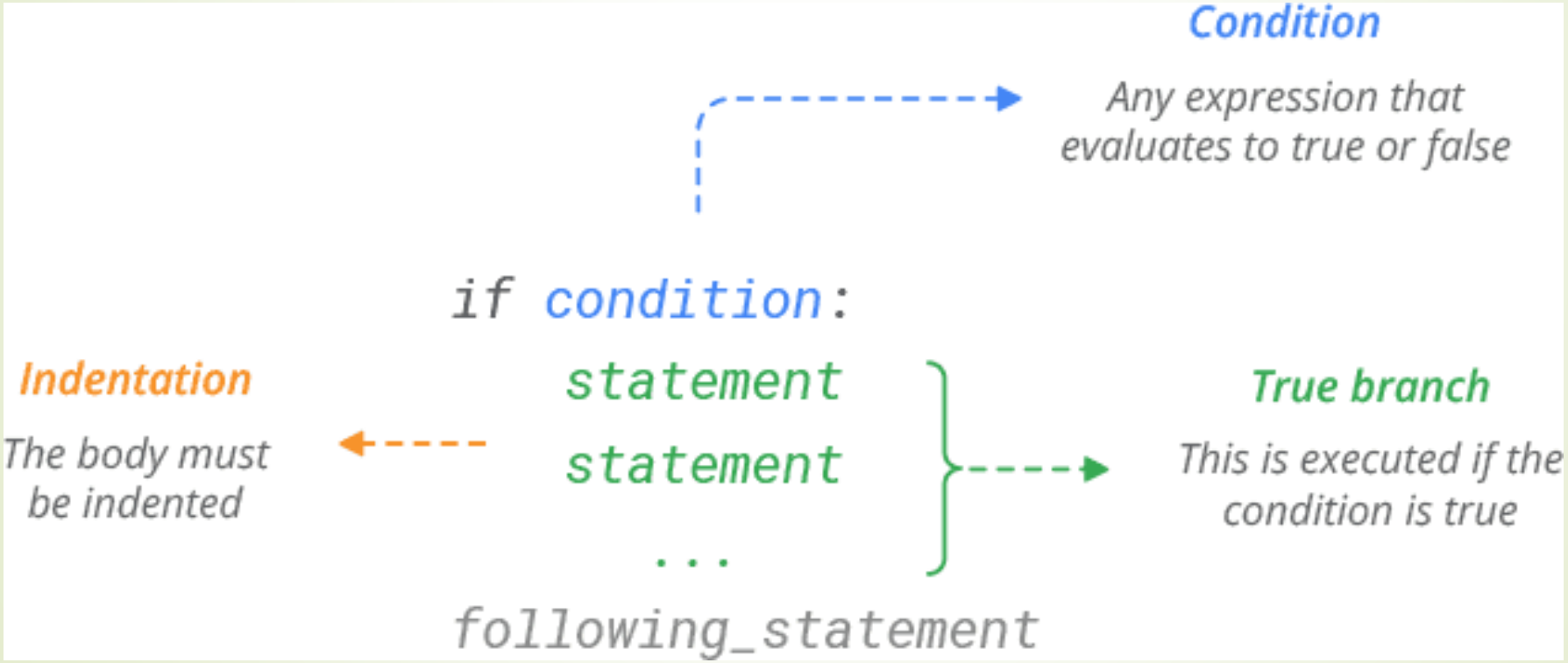
In Python, we generally have two loops/repetitive statements:

- for loop
- while loop

If statement

if Statement

Syntax :



if Statement

The program evaluates the test expression and will execute statement(s) only if the test expression is True.

If the test expression is False, the statement(s) is not executed.

In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.

Example – if statement

```
x, y= 7,5

if x>y:
   print("x is grater")
```

```
# Prints x is greater
```

Likewise, you can use the comparison operators to compare two values, as shown on the side.

Example
if x == y
if x != y
if x > y
if x >= y
if x < y
if x <= y

Why Indentation Syntax?

If you are used to a programming language that uses curly braces ({ and }) to delimit blocks of code, encountering blocks in Python for the first time can be disorienting, as Python doesn't use curly braces for this purpose.

Python uses indentation to mark a block of code, which Python programmers prefer to call **suite** as opposed to block (just to mix things up a little).

Suites

A colon introduces an indented suite of code

The colon (:) is important, in that it introduces a new suite of code that must be indented to the right. If you forget to indent your code after a colon, the interpreter raises an error.

```
x, y= 7,5

if x>y:
print("x is grater")
```

```
# Triggers SyntaxError: expected an indented block
```

Suites

Suites within any Python program are easy to spot, as they are always indented. This helps your brain quickly identify suites when reading code.

The other visual clue for you to look out for; is the colon character (:), which is used to introduce a suite that's associated with any of Python's control statements (such as if, else, for, and the like).

Why Indentation Syntax?

Nearly every programmer-friendly text editor has built-in support for Python's syntax model. In the IDLE Python GUI, for example, lines of code are automatically indented when you are typing a nested block; pressing the Backspace key backs up one level of indentation, and you can customize how far to the right IDLE indents statements in a nested block.

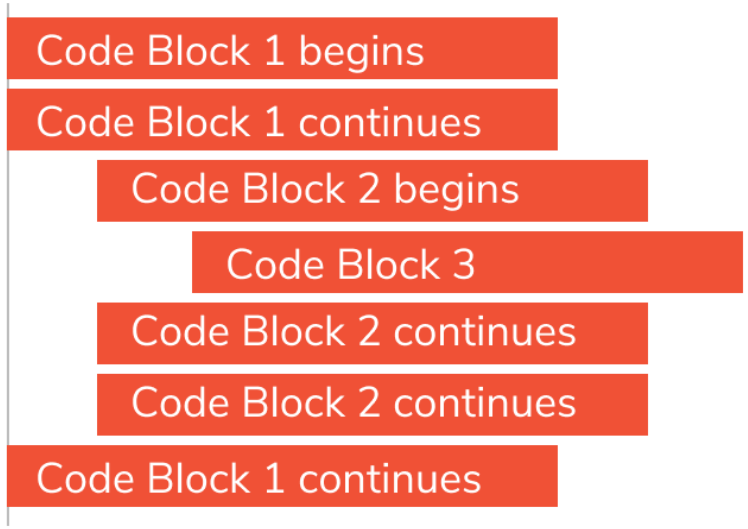
There is no universal standard on this: four spaces or one tab per level is common, but it's up to you to decide how and how much you wish to indent.

Indent further to the right, for further nested blocks, and less to close the prior block.

-> Indicates 1 Space Indendation

Statement 1
Statement 2
Statement 3
Statement 4
Statement 5
Statement 6
Statement 7

How the interpreter visualises



Here:

Statements 1, 2, 7 belong to code block 1 as they are at the same distance to the right.
Statements 3, 5, 6 belong to code block 2
Statement 4 belongs to code block 3

Execution happens in the same order.

! Don't Mix the Tabs and Spaces to Indent your code. It will create lot of confusion and very hard to debug also. So always stick to either Tabs or Spaces (Don't Mix tabs and spaces)

More Examples

In Python, any non-zero value or nonempty container is considered TRUE, whereas Zero, None, and empty container is considered FALSE. That's why all the below if statements are valid.

```
# any non-zero value
if -3:
    print('True')
# Prints True
```

```
# mathematical expression
x, y = 7, 5
if x + y:
    print('True')
# Prints True
```

```
# nonempty container
L = ['red','green']
if L:
    print('True')
# Prints True
```

```
if 0:
      print('True')
```

```
x,y= 7,5
if x-y:
      print("True")
```

```
L=['red','green']

if L:
      print("True")
```

Negating the expression

Consider this example:

```
number = int(input('Enter a number:'))

if number > 5:
       print(number,'is gratert than 5.')
```

To negate the conditional expression, use the logical not operator:

```
number = int(input('Enter a number:'))

if not number > 5:
    print(number,'is not gratert than 5.')
```

If statement and membership operators

You can also use the in keyword to check if a value is present in an iterable (string, list, tuple , dictionary, etc..):

```
s = 'linuxize'

if 'ze' in s:
       print('True.')
```

```
d= {'a': 2, 'b': 4}

if 'a' in d:
       print("True.")
```

When used on a dictionary, the in keyword checks whether the dictionary has a specific key.

One Line if Statement

Python allows us to write an entire if statement on one line.

```
#Short Hand If - single statement

x,y= 7,5
if x > y: print('x is grater')
```

```
# Prints x is greater
```

One Line if Statement

You can even keep several lines of code on just one line, simply by separating them with a semicolon ;

This is the only place in Python where semicolons are required: as statement separators.

```
#Short Hand If - multiple statement

x,y = 7,5
if x> y: print("x is greater"); print('x is greater'); print("x and y are not equal")
```

```
# Prints x is greater
# Prints y is smaller
# Prints x and y are not equal
```

Wrapping a large expression

Anywhere you need to code a large expression, simply wrap it in parentheses to continue it on the next line:

```
if (A == 1 and
    B == 2 and
    C == 3):
        print('spam' * 3)
```

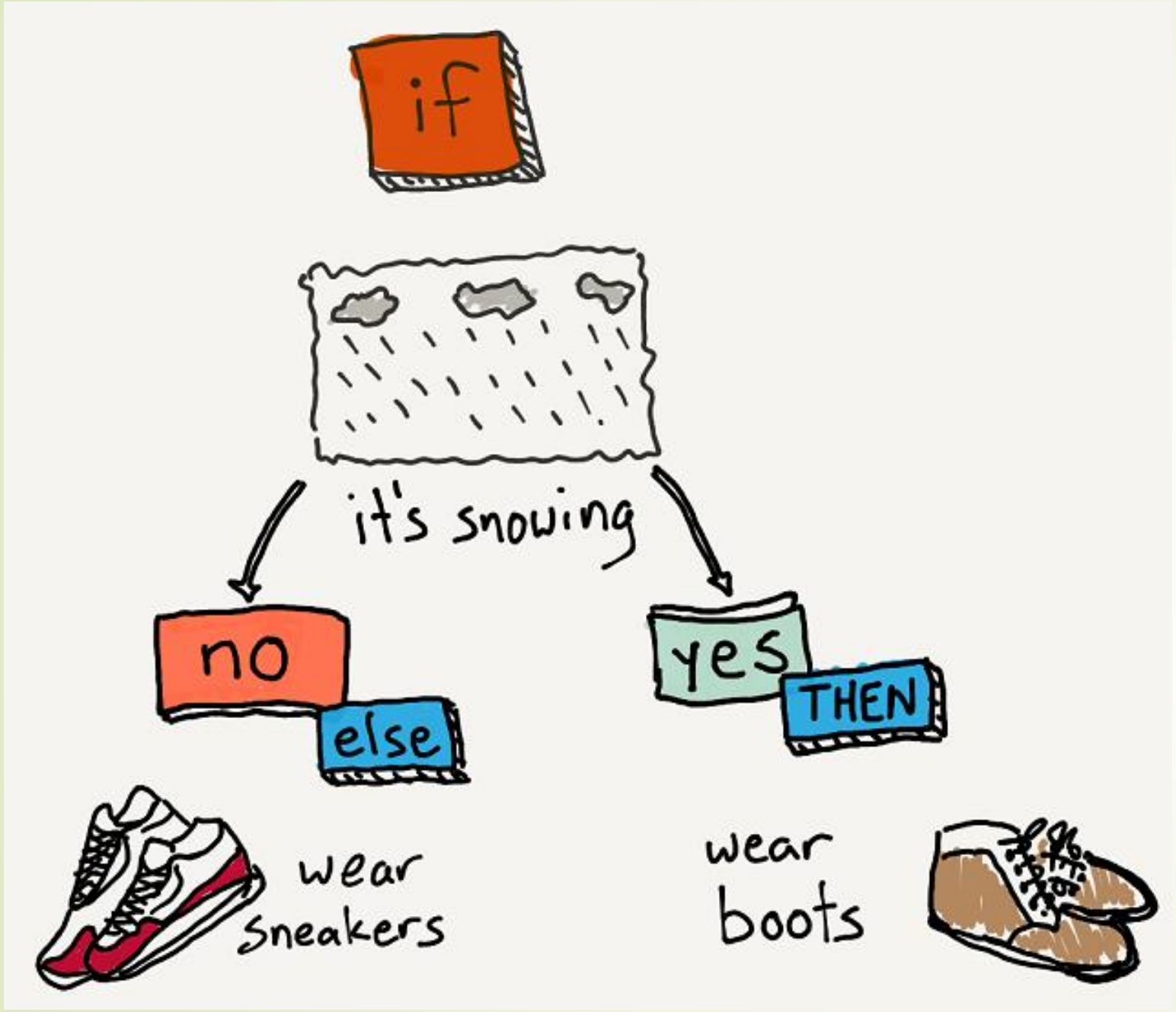
```
A,B,C= 1,2,3

if (A==1 and
    B==2 and
    C==3):
    print("Spam" *3)
```

Nested if Statement

You can nest statements within a code block

```
x,y,z = 7,4,2

if x>y:
        print("x is greater than y")
        if x>z:
                print("x is greater than y and z")
```

```
# Prints x is greater than y
# Prints x is greater than y and z
```



If – else statement

if…else Statement

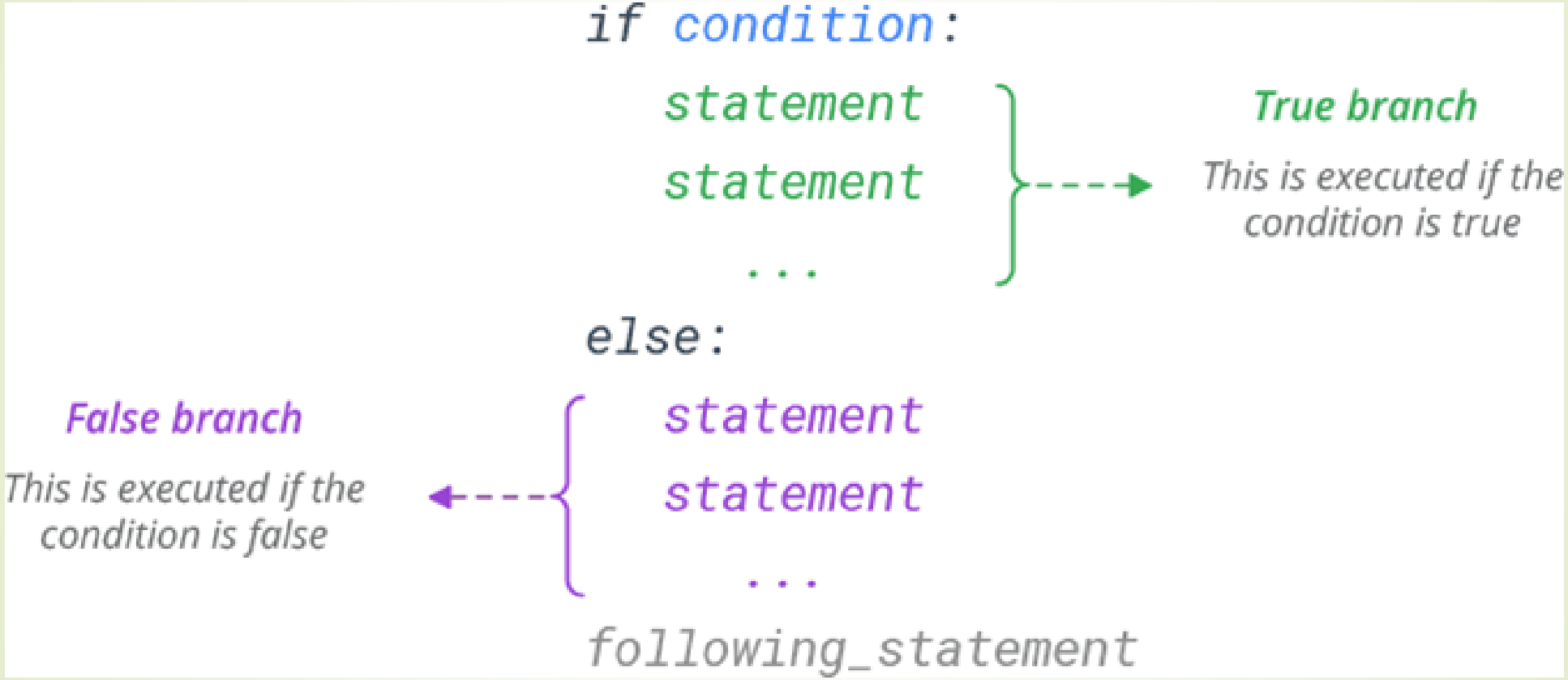
An if...else statement evaluates a condition and executes one of the two statements depending on the result.

The Python if...else statement takes the form shown on the next slide:

The if…else statement evaluates test expression and will execute the body of if only when the test condition is True.

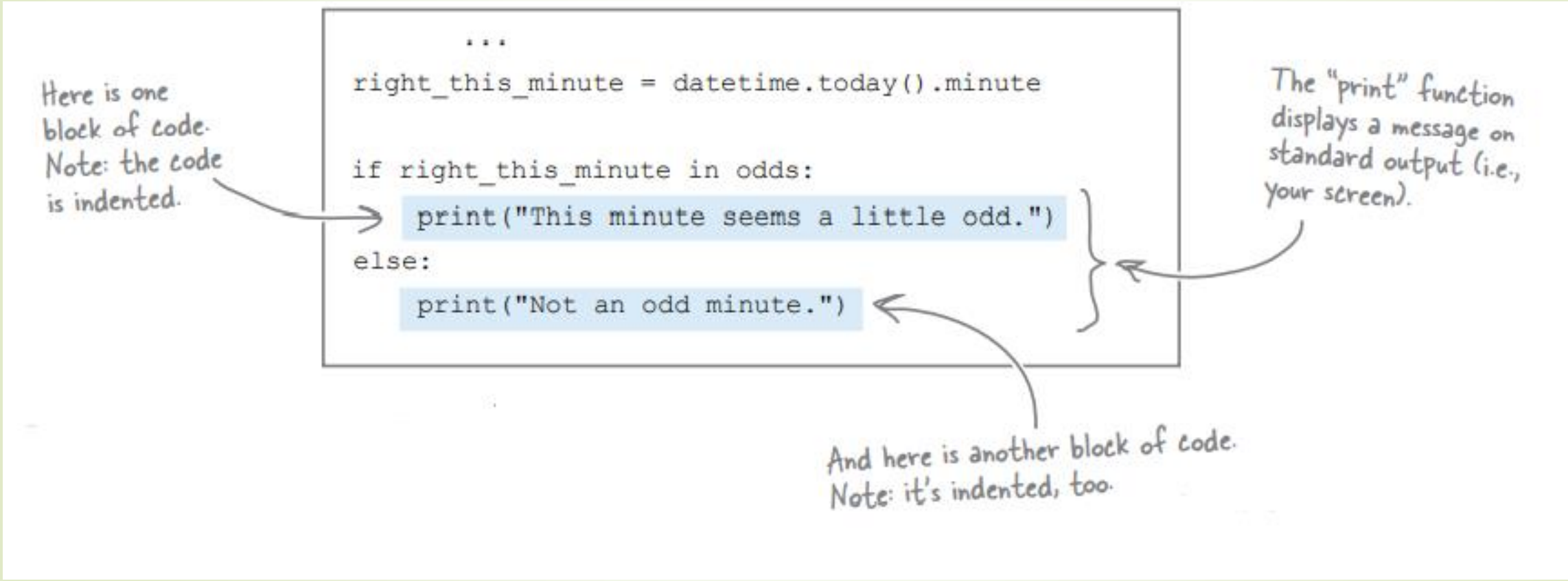
If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

Syntax – if-else



Example

The else keyword must end with (:) colon and to be at the same indentation level as the corresponding if keyword.



if…else Statement

Let's add an else clause to the previous example script:

```
number = int(input('Enter a number:'))

if number > 5:
	print(number,'is gratert than 5. ')
else:
	print(number, 'is qual or less than 5.')
```

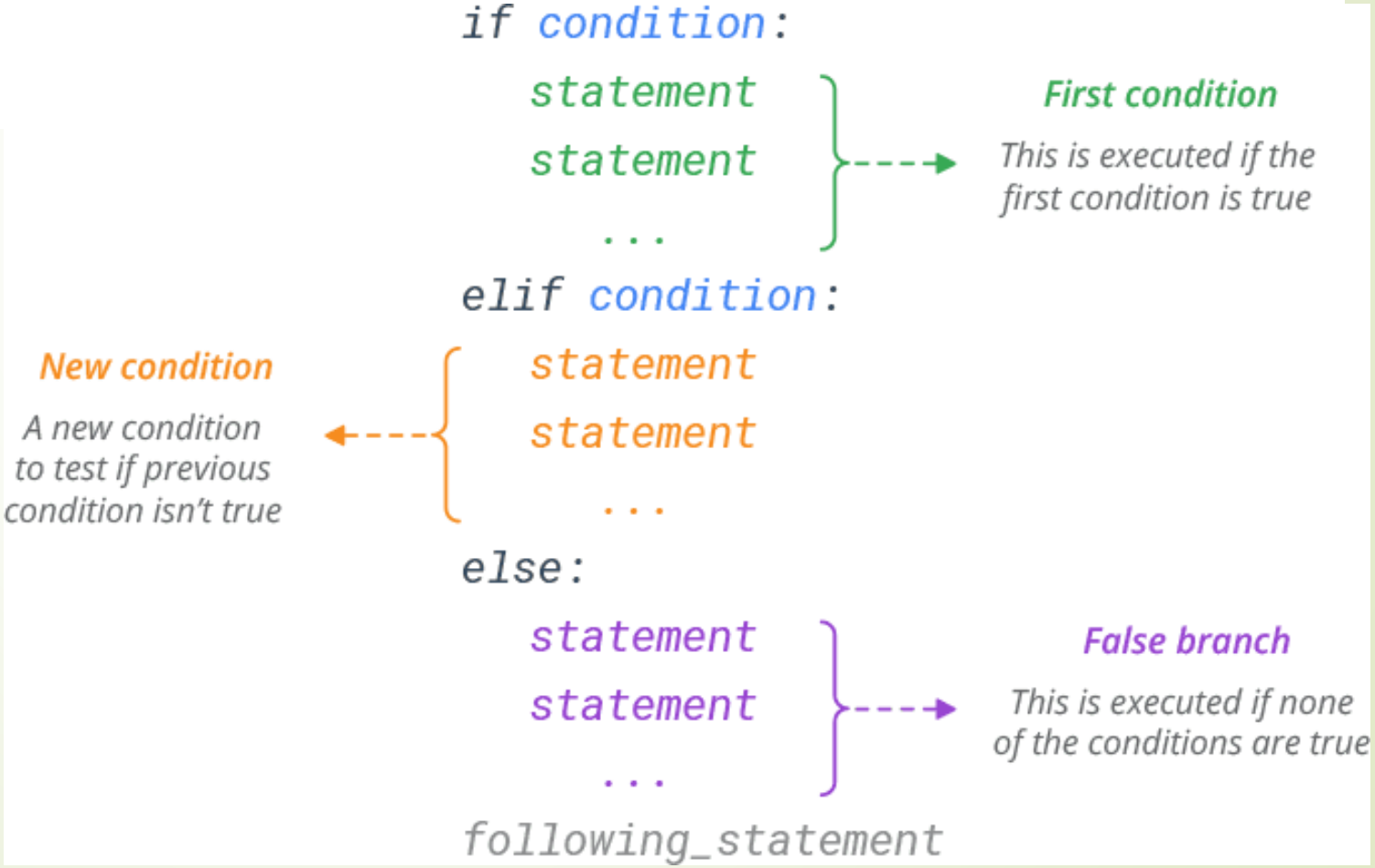
If you run the code and enter a number, the script will print a different message based on whether the number is greater or less/equal to 5.

The elif (else if) Statement

if..elif..else Statement

The elif keyword is short for else if.

Syntax



if..elif..else Statement

Let's add an elif clause to the previous script:

```
number = int(input('Enter a number:'))

if number > 5:
	print(number,'is gratert than 5.')
elif number < 5:
	print(number, ' less than 5.')
else:
	print(number, 'is equal to 5')
```

if..elif..else Statement

The elif allows us to check for multiple expressions.

If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, the body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

Substitute for Switch Case

Unlike other programming languages, Python does not have a ‘switch‘ statement. You can use if…elif…elif sequence as a substitute.

```
choice = int(input('choice 1-4:'))

if choice == 1:
      print("case 1")
elif choice == 2:
      print("case 2")
elif choice == 3:
      print("case 3")
elif choice == 4:
      print("case 4")
else:
      print("default case")
```

Nested if-else Statements

Python allows you to nest if statements within if statements.

Generally, you should always avoid excessive indentation and try to use elif instead of nesting if statements

The script shown on right, will prompt you to enter three numbers and will print the largest number among the numbers.

```
number1= int(input('Enter the frist numer:'))
number2= int(input('Enter the second numer:'))
number3= int(input('Enter the third numer:'))

if number1 > number2:
        if number1 > number3:
                print(number1, 'is the largest number.')
        else:
                print(number3, 'is the largest number.')
else:
        if number2 > number3:
                print(number2, 'is the largest number.')
        else:
                print(number3, 'is the largest number.')
```

Output

```
Enter the first number: 455
Enter the second number: 567
Enter the third number: 354
567 is the largest number.
```

Multiple Conditions

The logical or/and the and operators allow you to combine multiple conditions in the if statements.

Here is another version of the script to print the largest number among the three numbers. In this version, instead of the nested if statements, we will use the logical and operator and elif.

```
number1= int(input('Enter the frist numer:'))
number2= int(input('Enter the second numer:'))
number3= int(input('Enter the third numer:'))

if number1 > number2 and number1 > number3:
      print(number1, 'is the largest number.')
elif number2 > number3 and number2 > number1:
      print(number2, 'is the largest number.')
else:
      print(number3, 'is the largest number.')
```

While loop

A Python while loop behaves quite similarly to common English usage. If I say

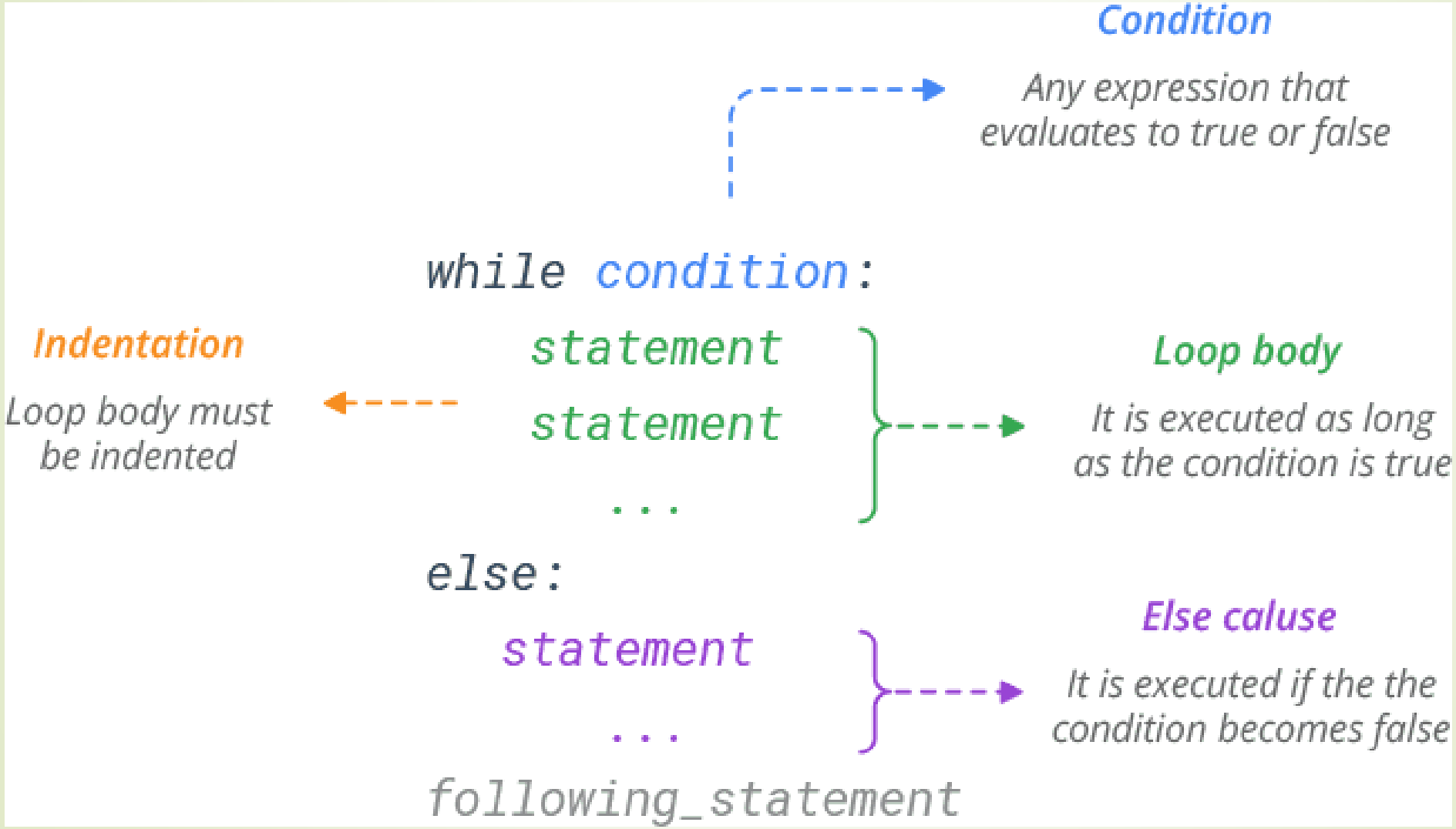
while your tea is too hot, add a chip of ice.

Presumably, you would test your tea. If it were too hot, you would add a little ice. If you test again and it is still too hot, you will add ice again. This process repeats until the tea gets to the right temperature.

Setting up the English example in a Python while format would look like this:

```
while your tea is too hot :
        add a chip of ice
```

While loop syntax

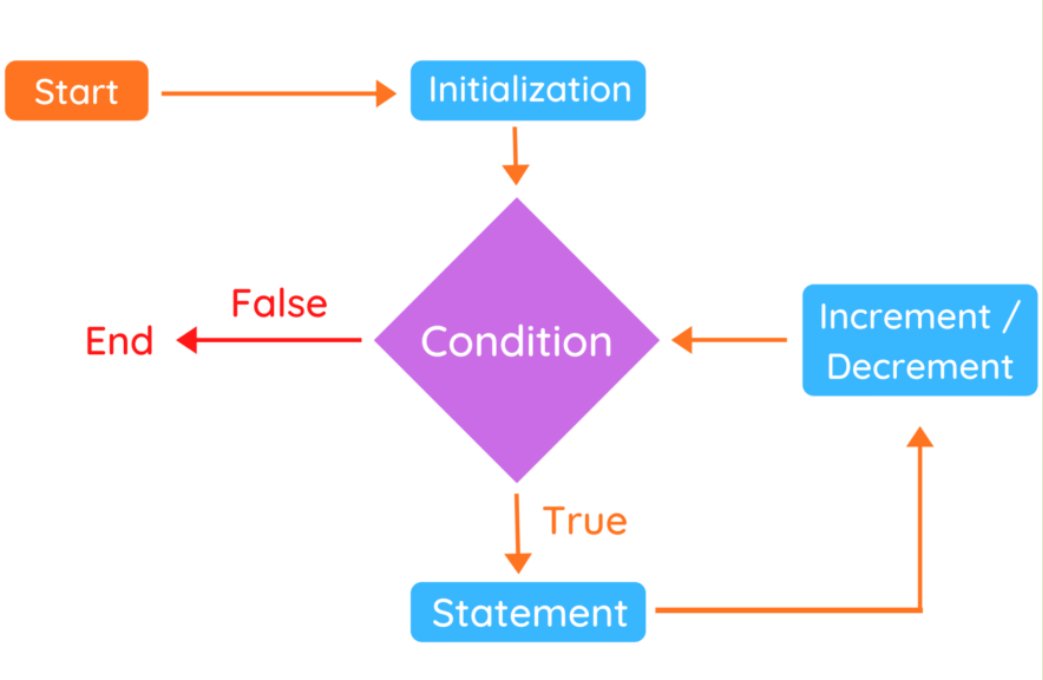


While loop

A while loop is a condition-controlled loop. One key thing to be noted is that the while loop is entry controlled, which means the loop can never run and the while loop is skipped if the initial test returns FALSE.

We generally use this loop when we don't know the number of times to iterate beforehand.

Python interprets any non-zero value as True. None and 0 are interpreted as False



While loop-Example

The tea starts at 115 degrees Fahrenheit. You want it at 112 degrees. A chip of ice turns out to lower the temperature one degree each time. You test the temperature each time, and also print out the temperature before reducing the temperature.

```
1  temperature = 115
2  while temperature > 112: #first while loop code
3         print(temperature)
4         temperature = temperature -1
5         print('....')
6
   print("the tea is cool enough.")
```

Python while loop: Example

```
#program to display 1 to 9

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

This program will initially check if the value of i is less than 10 or not. If it is TRUE, then it will print the value of i and the value of i will be increased by 1. This process will be repeated while the value of i is less than 10 i.e., 9.

While loop - Example

Here's another while loop involving a list, rather than a numeric comparison:

```
a= ['foo','bar','baz']
while a:
      print(a.pop(-1))
```

```
baz
bar
foo
```

When a list is evaluated in Boolean context, it is truthy if it has elements in it and falsie if it is empty.

In this example, a is true as long as it has elements in it. Once all the items have been removed with the .pop() method and the list is empty, a is false, and the loop terminates.

Exit condition as false

If the condition is false at the start, the while loop will never be executed at all.

```
# Exit condition is false at the start

x = 0
while x:
        print(x)
        x -=1
```

The infinite while loop

While the loop is skipped if the initial test returns FALSE, it is also forever repeated infinitely if the expression always returns TRUE.

For example, while loop in the following code will never exit out of the loop and the while loop will iterate forever.

```
# Infinite loop with while statement

while True:
      print("Press Ctrl+C to stop me!")
```

Interactive while Loops - Example

Suppose you want to let a user enter a sequence of lines of text and want to remember each line in a list.

The user may want to enter a bunch of lines and not count them all ahead of time. This means the number of repetitions would not be known ahead of time. A while loop is appropriate here.

There is still the question of how to test whether the user wants to continue. An obvious but verbose way to do this is to ask before every line if the user wants to continue, as shown below and in the example in the next slide.

Interactive while Loops

```
lines = list()
testAnswer = input("Press Y if you want to enter more lines: ")
while testAnswer == 'y':
	line = input("Next line:")
	lines.append(line)
	testAnswer = input("Press Y if you want to enter more lines: ")

print('Your lines were:')
for line in lines:
	print(line)
```

See the two statements setting testAnswer: one before the while loop and one at the bottom of the loop body.

Note : The data must be initialized before the loop, for the first test of the while condition to work. Also, the test must work when you loop back from the end of the loop body. This means the data for the test must also be set up a second time, in the loop body (commonly as the action in the last line of the loop). It is easy to forget the second time!

Interactive while Loops

The code works, but two lines must be entered for every one line you actually want!

A practical alternative is to use a sentinel: a piece of data that would not make sense in the regular sequence, and which is used to indicate the end of the input.

You could agree to use the line DONE! Even simpler: if you assume all the real lines of data will actually have some text on them, use an empty line as a sentinel.

This way you only need to enter one extra (very simple) line, no matter how many lines of real data you have.

Interactive while Loops

```
lines = list()
print("Enter lines of text.")
print("Enter an empty line to quit.")
line = input("Next line: ")          #initalize before the loop

while line != "":                    #while Not the terminatiom condition
      lines.append(line)
      line = input("Next line :")    # !! reset values at of loop!

print("your lines were:")
for line in lines:
      print(line)
```

Again, the data for the test in the while loop heading must be initialized before the first time the while statement is executed and the test data must also be made ready inside the loop for the test after the body has executed. Hence you see the statements setting the variable line both before the loop and at the end of the loop body. It is easy to forget the second place inside the loop!

After reading the rest of this paragraph, comment the last line of the loop out, and run it again: It will never stop! The variable line will forever have the initial value you gave it! You actually can stop the program by entering Ctrl-C.

While loop with else

While loops can also have an optional else block.

The else clause will be executed when the loop terminates normally (the condition becomes false).

The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

While loop with else - Examples

```
x= 6
while x:
        print(x)
        x-=1
else:
        print('Done!')
```

```
# Prints 6 5 4 3 2 1
# Prints Done!
```

The `else` clause will still be executed if the condition is false at the start.

```
x= 6
while x:
        print(x)
        x-=1
        if x==3:
                break
else:
        print('Done!')
```

```
# Prints 6 5 4
```

If the loop terminates prematurely with break, the else clause won't be executed.

```
x= 0
while x:
        print(x)
        x-=1
else:
        print('Done!')
```

```
# Prints Done!
```

The range() function

Definition and Usage – range()

The range() is used when a user needs to perform an action for a specific number of times.

range() is commonly used in *for* looping hence, knowledge of range is a key aspect when dealing with any kind of Python code. The most common use of range() function in Python is to iterate sequence type (List, string etc..) with for and while loop.

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

How to use range() function

Syntax

```
range(start, stop[, step])
```

It takes three arguments. Out of the three, two are optional. The start and step are optional arguments, and the stop is the mandatory argument.

Parameters of range()

start: (Lower limit) It is the starting position of the sequence. The default value is 0 if not specified. For example, range(0, 10). Here, start=0 and stop = 10 (*9)

stop: (Upper limit) generate numbers up to this number, i.e., An integer number specifying at which position to stop (upper limit). The range() never includes the stop number in its result.

step: Specify the increment value. Each next number in the sequence is generated by adding the step value to a preceding number. The default value is 1 if not specified. It is nothing but a difference between each number in the result. For example, range(0, 6, 1). Here, step = 1.

range(stop)

When you pass only one argument to the range(), it will generate a sequence of integers starting from 0 to stop-1.

```
#Print first 10 numbers
# stop = 10

for i in range (10):
       print(i, end=' ')
```

```
# Output 0 1 2 3 4 5 6 7 8 9
```

range(start, stop)

When you pass two arguments to the range(), it will generate integers starting from the start number to stop -1.

```
# Numbers from 10 to 15
# start = 15  stop = 16

for i in range(10, 16):
       print (i, end=' ')
```

```
# Output 10 11 12 13 14 15
```

range(start, stop, step)

When you pass all three arguments to the range(), it will return a sequence of numbers, starting from the start number, increments by step number, and stops before a stop number.

Here you can specify a different increment by adding a step parameter.

```
#Numbers from 10 to 50 counting in 5
#start = 10 stop = 50  step = 5

for i in range(10, 50, 5):
      print (i, end=' ')
```

```
# Output 10 15 20 25 30 35 40 45
```

How to use Python **range(start, stop, step)**

range() returns the immutable sequence of numbers starting from the given **start integer to the stop-step**. Each number is incremented by adding step value to its preceding number

range(0, 6, 1) → | 0 | 1 | 2 | 3 | 4 | 5 |

Step. (Optional) Specify the increment. Default is **1**

Stop. (Required) specifying at which position to stop. Not part of the result

Start. (Optional) Start number of sequence. Default is **0**

```
for i in range(6):
    print(i)
```

It returns a range object not list

type(range(6)) -> class 'range'

Reverse range

You can display the sequence of numbers produced by a range() function by descending order or reverse order.

You can use the following two ways to get the reverse range of numbers.

- Use a negative step value
- Use a reversed() function

range(1, 6, 1) ← Normal

1 range(5, 0, -1) ← Reverse

2 reversed(range(1, 6, 1))


```
a= list( range(1,6,1) )
b= list( range(5,0,-1))
c= list( reversed(range(1,6,1)))

print(a,b,c, sep="\n")
```

range()

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

```
print(range(10))
print(list(range(10)))
print(list(range(2,8)))
print(list(range(2, 20,3)))
```

Output

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

Access range() attributes

It is essential to know the range() attributes when you receive it as input to your function, and you wanted to see the value of the start, stop and step argument.

```
range_1 = range(0, 10)
print(range_1)

# access range() attributes
print (range_1.start) # 0
print (range_1.stop) # 10
print (range_1.step) # 1
```

```
>>> range1 = range(2,2)
>>> range1.start
2
>>> range1.stop
2
>>> range1.step
1
>>> list(range1)
[]
>>>
```

range() indexing and slicing

Built-in function range() is the constructor that returns a range object, this range object can also be accessed by its index number using indexing and slicing.

Indexing

range() supports both positive and negative indices. The example demonstrates the same.

In the case of range(), The index value starts from zero to (stop). For example, if you want to access the 3rd number, we need to use 2 as the index number.

The numbers can be accessed from right to left by using negative indexing.

```
range1 = range(0, 10)

# first number (start number) in range
print (range1[0])

#access 5th number in range
print (range1[5])

#access last number
print (range1 [range1.stop - 1])
```

```
#negative indexing
# access last number
print (range(10)[-1])

#access second last number
print (range (10)[-2])
```

Slicing

Slicing implies accessing a portion from range()

```
# slicing
for i in range (10)[3:8]:
       print (i, end=' ')
```

Points to remember

The range() function only works with the integers, So all arguments must be integers. You cannot use float numbers or any other data type as a start, stop, and step value.

All three arguments can be positive or negative.

The step value must not be zero. If a step=0, Python will raise a ValueError exception.

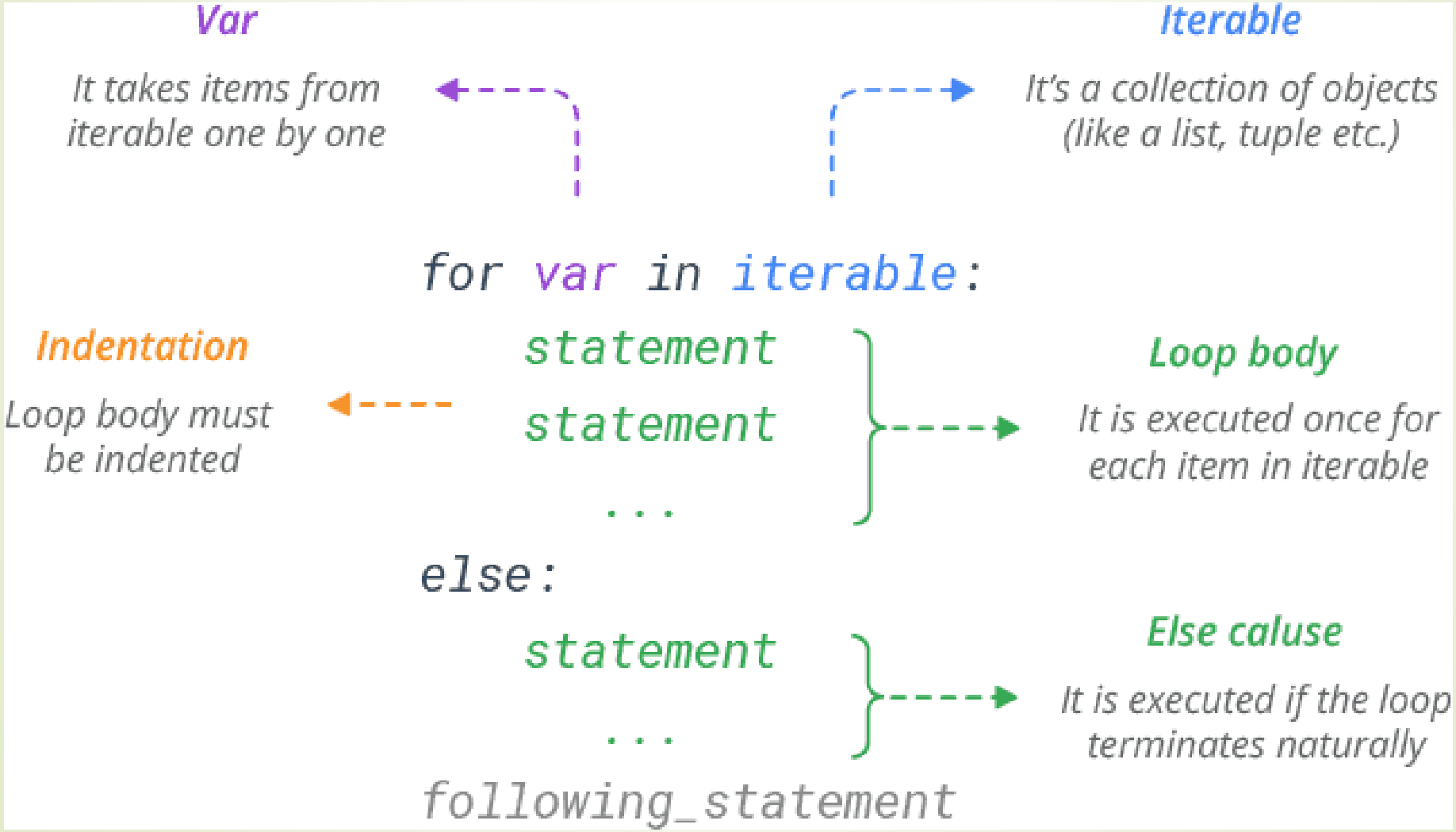
The for loop

- A for loop is a programming concept that, when it's implemented, executes a piece of code repeatedly "for" a certain number of times, based on a sequence.

- In contrast to the while loop, there isn't any condition actively involved - you just execute a piece of code repeatedly for a number of times.

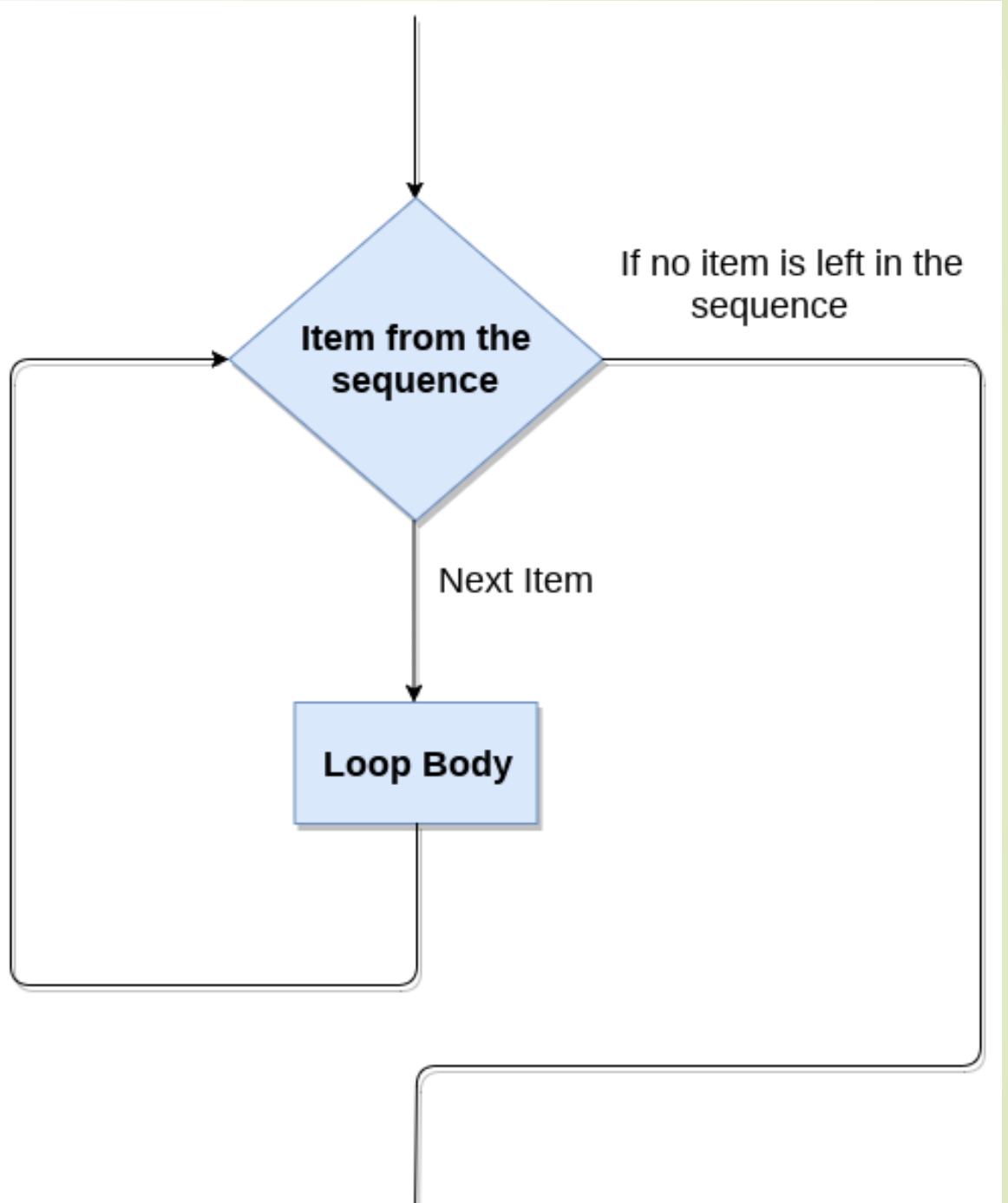
- In other words, while the while loop keeps on executing the block of code contained within it only when the condition is True, the for loop executes the code contained within it only for a specific number of times. This "number of times" is determined by a sequence or an ordered list of things.

for statement



The for Statement

- Here, var is the variable that takes the value of the item inside the sequence on each iteration.
- Loop continues until we reach the last item in the sequence or iterable.
- The body of for loop is separated from the rest of the code using indentation.



Example – Iterating over a List

```
# Program to find the sum of all number stored in a list

# List of number
numbers = [6,5,3,8,4,2,5,4,11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
       sum = sum+val

print ("the sum is", sum)
```

```
The sum is 48
```

Iterating over a Tuple

In the following program, for loop will iterate over the tuple of first four prime numbers and print them one by one.

```
#Tuple of items
tuple_prime = (2,3,5,7)

print ('These are the frist four prime number')

#Iterating over the tuple
for item in tuple_prime:
      print (item)
```

Output

```
These are the first four prime numbers
2
3
5
7
```

Iterate through a string

```
# Iterate through a string
S = 'python'

for x in S:
       print(x,end =' ' )

# Prints p y t h o n
```

Iterating using the range function

If you need to execute a group of statements for a specified number of times, use built-in function range().

```
# print 'Hello' three times
for x in range(3):
        print ('Hello!')
```

```
# Prints Hello!
# Prints Hello!
# Prints Hello!
```

```
# generate a sequence of number from 2 to 6
for x in range (2, 7):
        print (x)
```

```
# Prints 2 3 4 5 6
```

```
for x in range (-5,0):
        print(x)
```

```
# Prints -5 -4 -3 -2 -1
```

```
# Increment the range with 2
for x in range (2, 7, 2):
        print(x)
```

```
# Prints 2 4 6
```

```
# Python range function example

for i in range(15, 110, 10):
      print(i, end = ', ')

print('\n')
for i in range(150, 1, -10):
      print(i, end = ', ')

print('\n')
for i in range(-150, -300, -25):
      print(i, end = ', ')

print('\n')
for i in range(-300, -150, 25):
      print(i, end = ', ')

print('\n')
for i in range(-100, -500, 25):
      print(i, end = ', ')

print('\n')
for i in range(400, -200, -50):
      print(i, end = ', ')
```

```
15, 25, 35, 45, 55, 65, 75, 85, 95, 105,

150, 140, 130, 120, 110, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10,

-150, -175, -200, -225, -250, -275,

-300, -275, -250, -225, -200, -175,

-100, -50, 0, 50, 100, 150, 200, 250, 300, 350, 400, 450,

400, 350, 300, 250, 200, 150, 100, 50, 0, -50, -100, -150,
>>>
```

Example

Let's look at a problem where we will make use of the loop variable. The program below prints a rectangle of stars that is 4 rows tall and 6 rows wide.

```
for i in range (4):
        print ('*'*6)
```

The rectangle produced by this code is shown below

```
******
******
******
******
```

Example

Suppose we want to make a triangle instead. Like this

```
for i in range (1,5):
        print ('* '*i)
```

We can accomplish this with a very small change to the rectangle program. Looking at the program, we can see that the for loop will repeat the print statement four times, making the shape four rows tall. It's the 6 that will need to change. What's the change that you will do?

If statement within a for loop

Inside a for loop, you can use if statements as well.

Let's look at an example of iterating over the list of integers and print if it's an even number or an odd number.

```
list_ints =[1, 2, 3, 4, 5]

for i in list_ints:
        if i % 2 == 0:
                print(f'{i} is even.')
        else:
                print(f'{i} is odd.')
```

```
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
```

Iterating over Lists with range()

We can access all the elements, in the list with a for loop, but the index of an element is not available.

However, there is a way to access both the index of an element and the element itself.

The solution lies in using range() in combination with the length function len() or using the index() function.

Access Index in for Loop

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
colors = ['red', 'green', 'blue']
for index in range(len(colors)):
        print(index, colors[index])
```

```
# Prints 0 red
# Prints 1 green
# Prints 2 blue
```

Or use the index() as shown below:

```
colors = ['red', 'green', 'blue']
for color in colors:
        print(color, colors.index(color))
```

```
red 0
green 1
blue 2
```

Iterate Through a Dictionary

Dictionaries are an useful and widely used data structure in Python.

As a Python coder, you'll often be in situations where you'll need to iterate through a dictionary in Python, while you perform some actions on its key-value pairs.

Iterate Through a Dictionary

As you can see below, when a for loop iterates through a dictionary, the loop variable is assigned to the dictionary's keys.

```
d = {'foo' : 1, 'bar' : 2, 'baz' : 3 }

for k in d:
        print(k)
```

```
...
foo
bar
baz
```

Iterate Through a Dictionary

To access the dictionary values within the loop, you can make a dictionary reference using the key as usual:

```
d = {'foo' : 1, 'bar' : 2, 'baz' : 3 }
for k in d:
        print (d[k])
```

```
1
2
3
```

You can also iterate through a dictionary's values directly by using .values():

```
d = {'foo' : 1, 'bar' : 2, 'baz' : 3 }

for v in d.values():
        print(v)
```

```
1
2
3
```

Iterate Through a Dictionary

In fact, you can iterate through both the keys and values of a dictionary simultaneously.

That is because the loop variable of a for loop isn't limited to just a single variable. It can also be a tuple, in which case the assignments are made from the items in the iterable using packing and unpacking, just as with an assignment statement:

```
i,j = (1, 2)
print(i, j)

for i, j in [(1, 2), (3, 4), (5, 6)]:
        print(i, j)
```

```
1 2
3 4
5 6
```

Iterate Through a Dictionary

As noted in the chapter on Python dictionaries, the dictionary method .items() effectively returns a list of key/value pairs as tuples:

```
d = {'foo': 1, 'bar': 2, 'baz': 3}
d.items()

#dict_items([('foo@, 1), ('bar', 2), ('baz', 3)])
```

Thus, the Pythonic way to iterate through a dictionary accessing both the keys and values looks like this:

```
d = {'foo': 1, 'bar' : 2, 'baz' : 3}

for k, v in d.items():
      print ('k =', k, ', v =', v)
```

```
k = foo , v = 1
k = bar , v = 2
k = baz , v = 3
```

Zip function

It is used when there are two or more iterable and they need to be aggregated into one iterable.

Syntax

```
zip(iterator1, iterator2, iterator3 ...)
```

Parameter Values

Parameter	Description
iterator1, iterator2, iterator3 ...	Iterator objects that will be joined together

Zip function

The function takes in iterable as arguments and returns an iterator. This iterator generates a series of tuples containing elements from each iterable.

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
result = zip(list1, list2)
print(result)
```

```
<zip object at 0x0000025329B6F7C0>
```

Here, you use zip(list1, list2) to create an iterator that produces tuples of the form (x, y). In this case, the x values are taken from list1 and the y values are taken from list2.

Notice how the Python zip() function returns an iterator. To retrieve the final list object, you need to use list() to consume the iterator.

Zip function

zip() can accept any type of iterable, such as files, lists, tuples, dictionaries, sets, and so on.

```
columns = ['employee_id', 'employee_name', 'employee_salary']
values = [1, 'John', '50000']

employee_details = list(zip(columns, values))

print(employee_details)
```

OUTPUT:

```
[('employee_id', 1), ('employee_name', 'John'), ('employee_salary', '50
```

Creating a dictionary using columns and values.

```
columns = ['employee_id', 'employee_name', 'employee_salary']

values = [1, 'Smith', 80000]

employee_details = dict(zip(columns, values))

print(employee_details)
```

OUTPUT:

```
{'employee_id': 1, 'employee_name': 'Smith', 'employee_salary': 80000}
```

Passing Arguments of Unequal Length

It's possible that the iterables you pass in as arguments aren't the same length.

In these cases, the number of elements that zip() puts out will be equal to the length of the shortest iterable.

The remaining elements in any longer iterables will be totally ignored by zip(),

```
list1 = [1,2,3,4,5]
tup1 = ('a','b','c')
result = list(zip(list1,tup1))
print(result)
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Looping Through Multiple Lists

Using built-in zip() function you can loop through multiple lists at once.

```
# Loop through two lists at once
name = ['Bob','Sam','Max']
age = [25, 35, 30]

for x, y in zip(name, age):
      print(x, y)
```

```
# Prints Bob 25
# Prints Sam 35
# Prints Max 30
```

Modify a List While Iterating

Don't alter mutable objects while looping on them. It may create an infinite loop.

```
#infinute loop
colors= ['red', 'green', 'blue']
for x in colors:
      if x == 'red':
            colors.insert(0, 'orange')
            print(colors)

print(colors)
```

It is recommended that you first make a copy. The slicing operator makes this especially convenient.

```
colors =['red', 'green', 'blue']

for x in colors[:]:
      if x=='red':
            colors.insert(0, 'orange')
            print('*')

print(colors)
```

enumerate

```
values = ["a", "b", "c"]

index = 0
for value in values:
	print(index, value)
	index += 1
```

```
 values = ["a", "b", "c"]

for count, value in enumerate(values):
	print(count, value)
```

Nested Loops

One loop inside another loop is called a Nested Loop. In Python, different nested loops can be formed using for and while loops - a for-in loop inside another for-in loop, a while loop inside another while loop, a for-in loop inside a while loop, a while inside the for-in loop.

Why do we need nested loops?

Similar to Loops in Python, nested loops execute a block of statements until the conditions return to be False. These loops are used when we need to execute a block of statements under multiple conditions. We can use these loops to traverse through multi-dimensional arrays, matrix values, tabular data.

Nested Loops

Syntax for nested `for` loop

```
for [first iterating variable] in [outer loop]: # Outer loop
    [do something]  # Optional
    for [second iterating variable] in [nested loop]:   # Nested loop
        [do something]
```

Syntax for nested while loop

```
while condition:
    while condition:
        statement (s)
```

Other Nested Loops like for inside while and while inside for, can be also written.

Nested for loop - example

Let's implement a nested for loop so we can take a closer look. In this example, the outer loop will iterate through a list of integers called num_list, and the inner loop will iterate through a list of strings called alpha_list.

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']

for number in num_list:
       print (',number)
       for letter in alpha_list:
              print(letter, end=' ')
```

```
Output
1
a
b
c
2
a
b
c
3
a
b
c
```

Nested while loop - Example

```
Outer loop (runs n times)
    Inner loop (runs m times)
        Body of inner loop (runs n x m times)
```

```
i=1
while (i <= 2):
       j = 1
       while (j <=3 ):
              print ("i = ", i, " j = ", j)
              j = j + 1
       i = i+1

print("Done!")
```

```
i =  1  j =  1
i =  1  j =  2
i =  1  j =  3
i =  2  j =  1
i =  2  j =  2
i =  2  j =  3
Done!
```

Examples

```
colors = ['yellow', 'black', 'green', 'purple']
fruits = ['mango', 'banana', 'apple']

for x in colors:
      for y in fruits:
            print(x,y)
```

```
yellow mango
yellow banana
yellow apple
black mango
black banana
black apple
green mango
green banana
green apple
purple mango
purple banana
purple apple
```

Nested for Loop

Nested for loops can be useful for iterating through items within lists composed of lists. In a list composed of lists, if we employ just one for loop, the program will output each internal list as an item:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0, 1, 2], [9.9, 8.8, 7.2]]

for list in list_of_lists:
      print(list)
```

```
Output
['hammerhead', 'great white', 'dogfish']
[0, 1, 2]
[9.9, 8.8, 7.7]
```

Nested for loop

In order to access each individual item of the internal lists, we'll implement a nested for loop:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0, 1, 2], [9.9, 8.8, 7.2]]

for list in list_of_lists:
        for item in list:
                print(item)
```

```
Output
hammerhead
great white
dogfish
0
1
2
9.9
8.8
7.7
```

Nested for Loops - Dictionaries

```
my_hash = {}

my_hash["Brazil"] = ["Sao Paulo", "Rio de Janeiro", "Recife"]
my_hash ["USA"] = ["San Diego", "New York", "Santa Barbara", "Miami"]
my_hash["Japan"] = ["Okinawa", "Tokyo", "Osaka"]

for country_name in my_hash:
	for city_name in my_hash[country_name]:
		print (country_name, city_name)
	print()
```

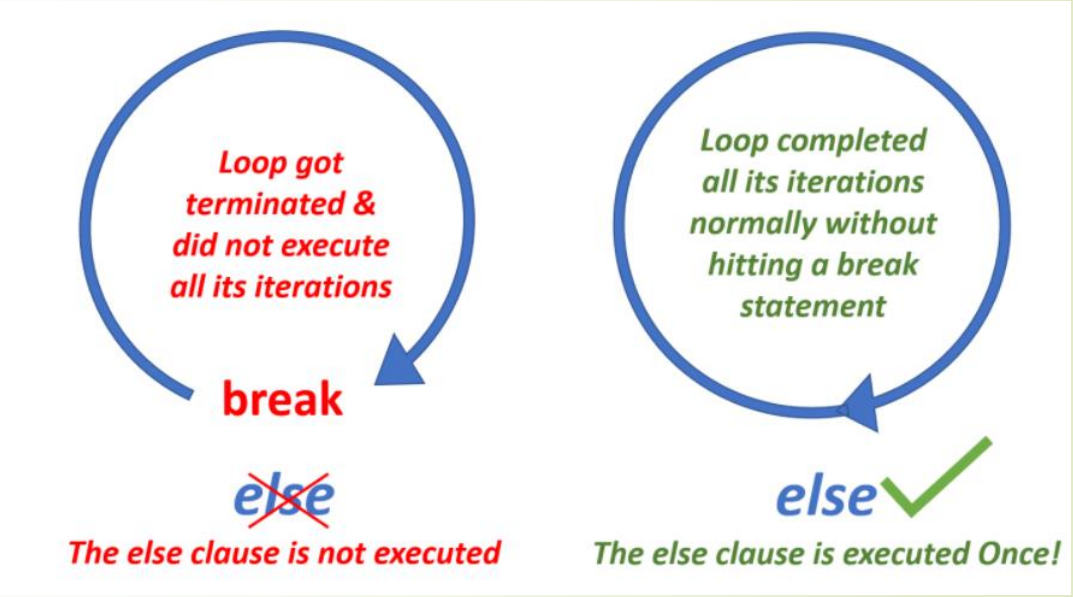
```
>>> Brazil Sao Paulo
>>> Brazil Rio de Janeiro
>>> Brazil Recife

>>> USA San Diego
>>> USA New York
>>> USA Santa Barbara
>>> USA Miami

>>> Japan Okinawa
>>> Japan Tokyo
>>> Japan Osaka
```

Understanding For-Else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The else clause will not be executed if the loop gets terminated by a break statement.

If a loop does not hit a break statement, then the else clause will be executed once after the loop has completed all its iterations (meaning, after the loop has completed normally).



Example – search for an item in the list

```
my_tresure_box = ["a", "b", "Diamond", "c"]

search_item = input("Enter the item to be searched:")
for item in my_tresure_box:
	if(item==search_item):
		print(f"{search_item} Found!")
		break
else:
	print(f"No{search_item} found.")
```

```
Enter the item to be searched:Diamond
Diamond Found!
```

```
Enter the item to be searched:Gold
No Gold found.
```