Flow Control

Program Control Flow

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

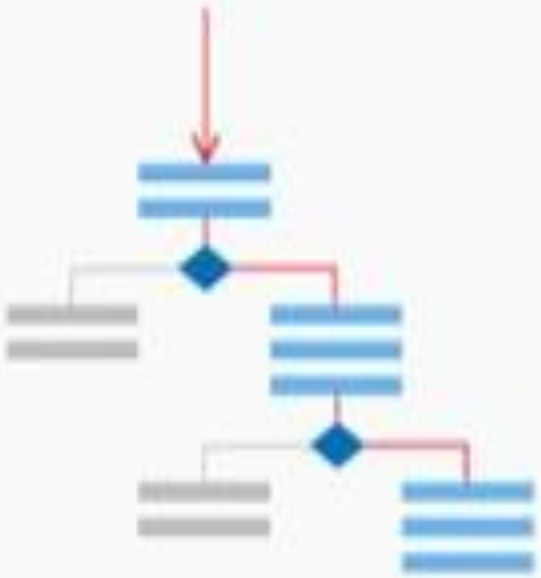
The control flow of a Python program is regulated by conditional statements, loops, and function calls. Raising and handling exceptions also affects control flow; exceptions will be covered in a later chapter.

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

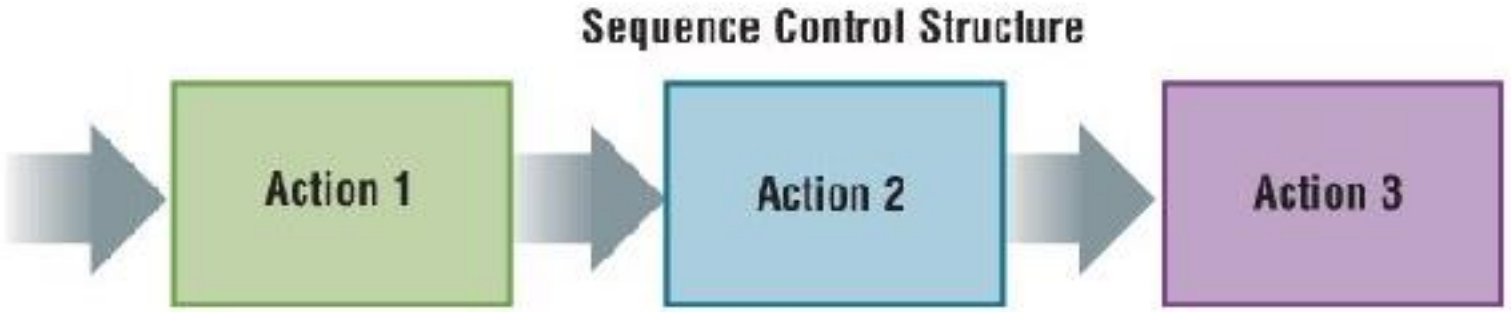
Certain statements will be executed repeatedly

1. 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;
```

2. 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

3. Repetition

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

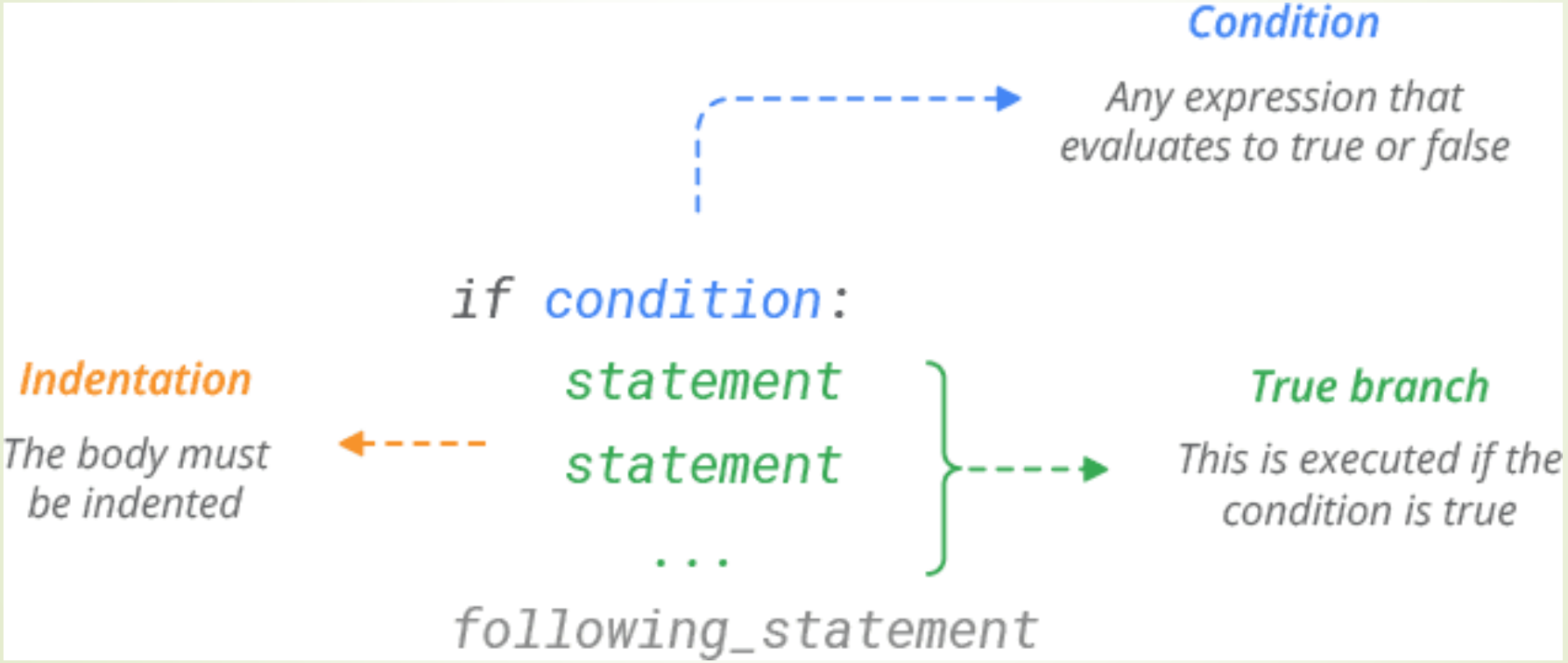
In Python, we generally have two loops/repetitive statements:

- 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.

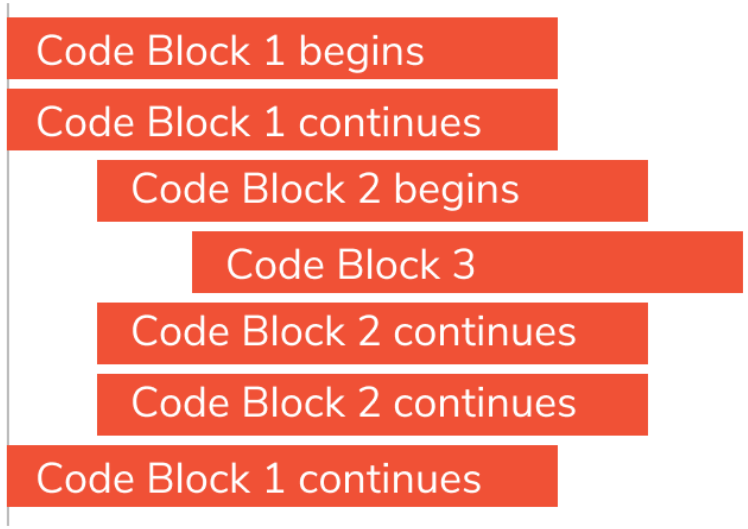
There is no universal standard on this: four spaces or one tab per level is common, but it's up to you to decide how and how much you wish to indent.

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
```

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

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

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

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

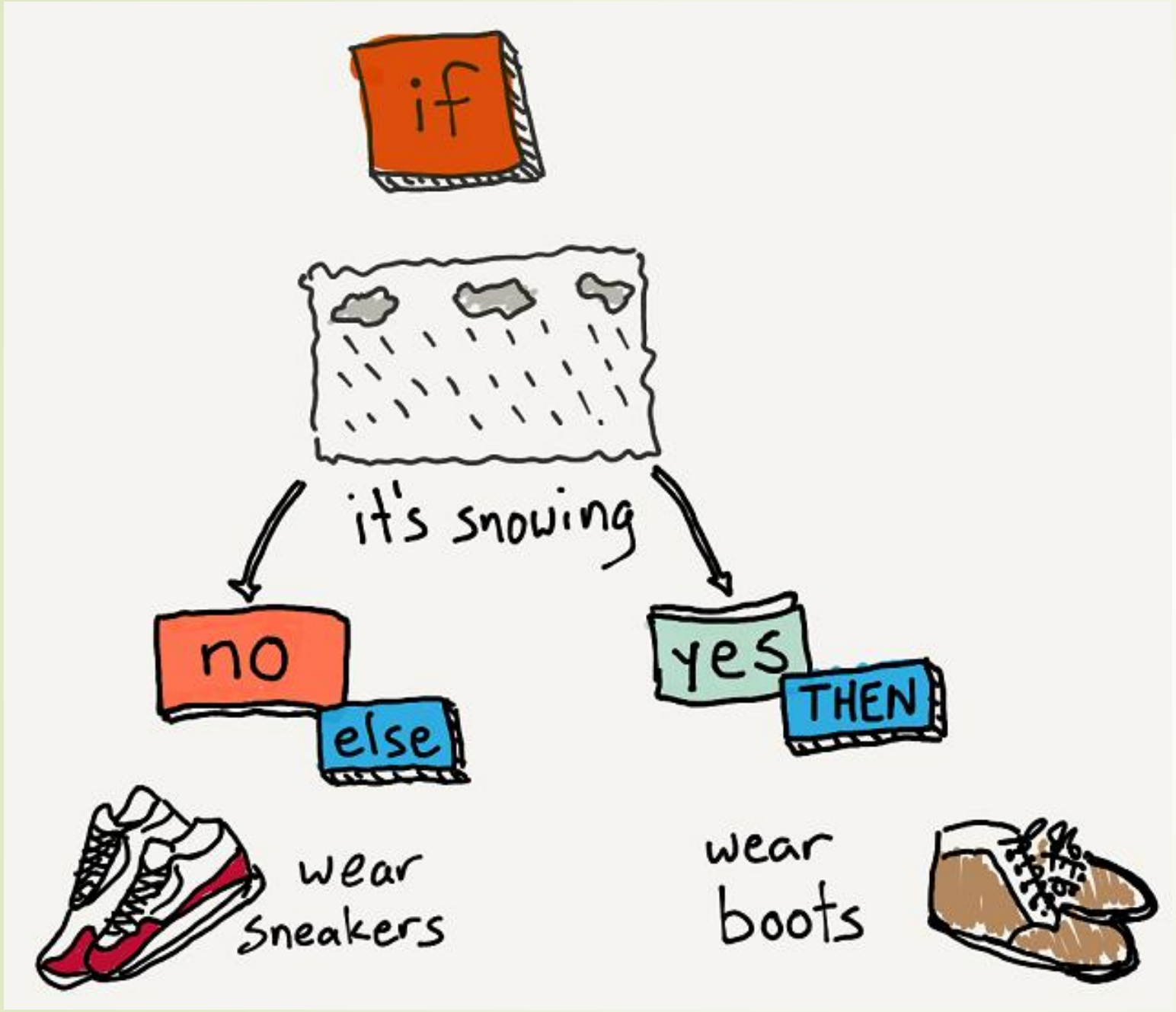
Nested if Statement

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

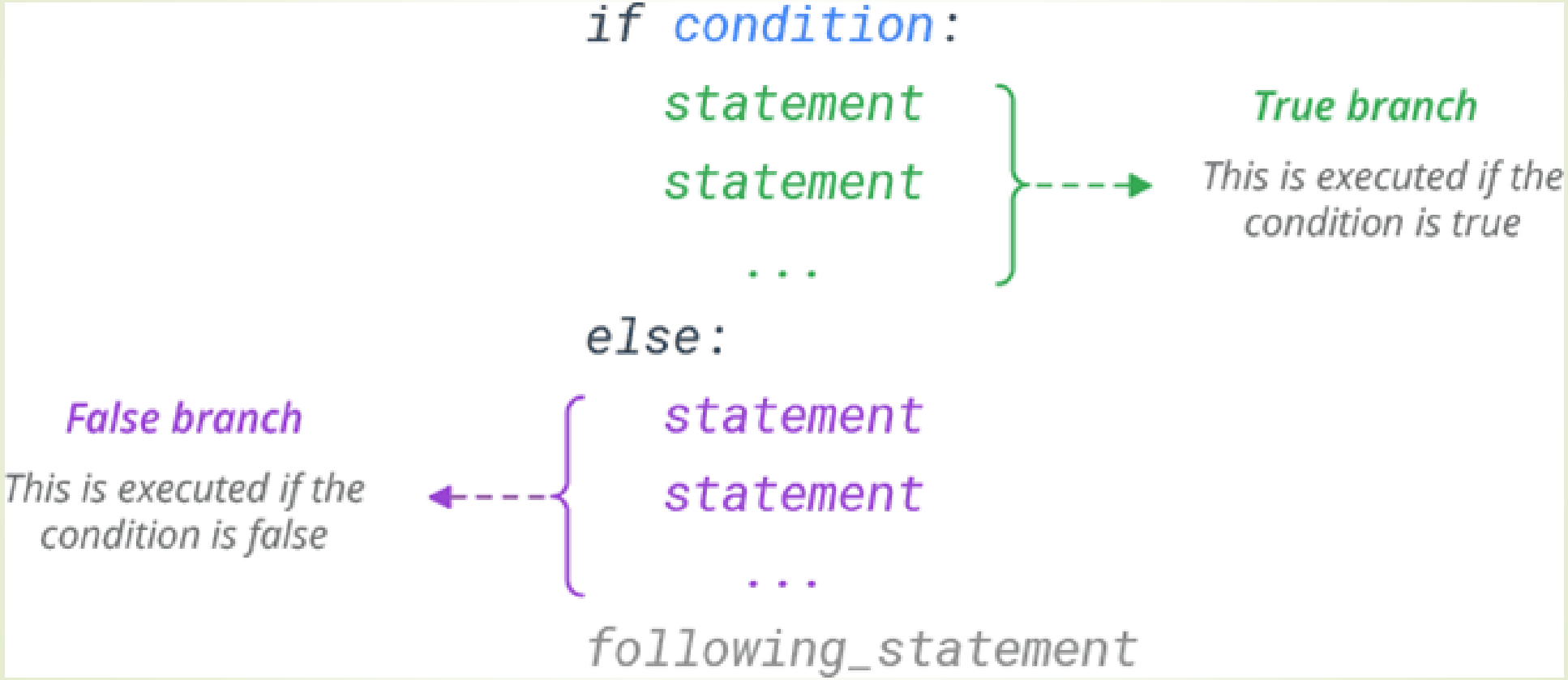
An if...else statement evaluates a condition and executes one of the two statements depending on the result.

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.

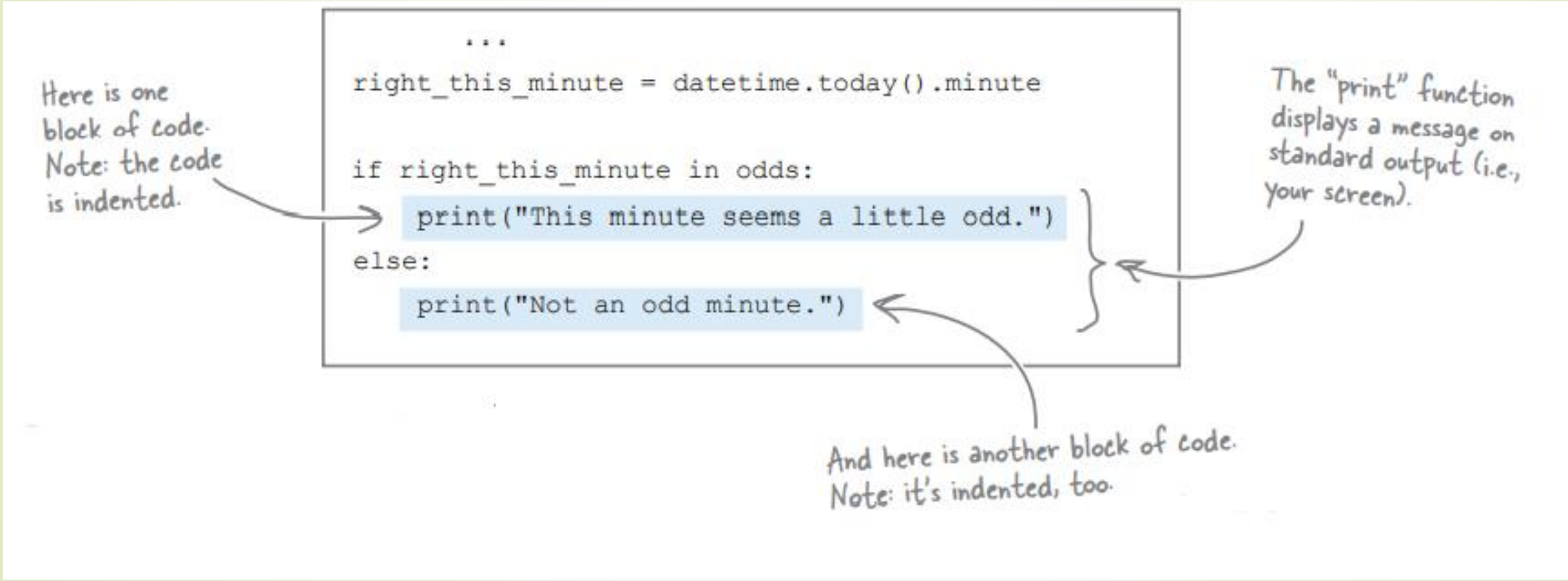
If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

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.')
```

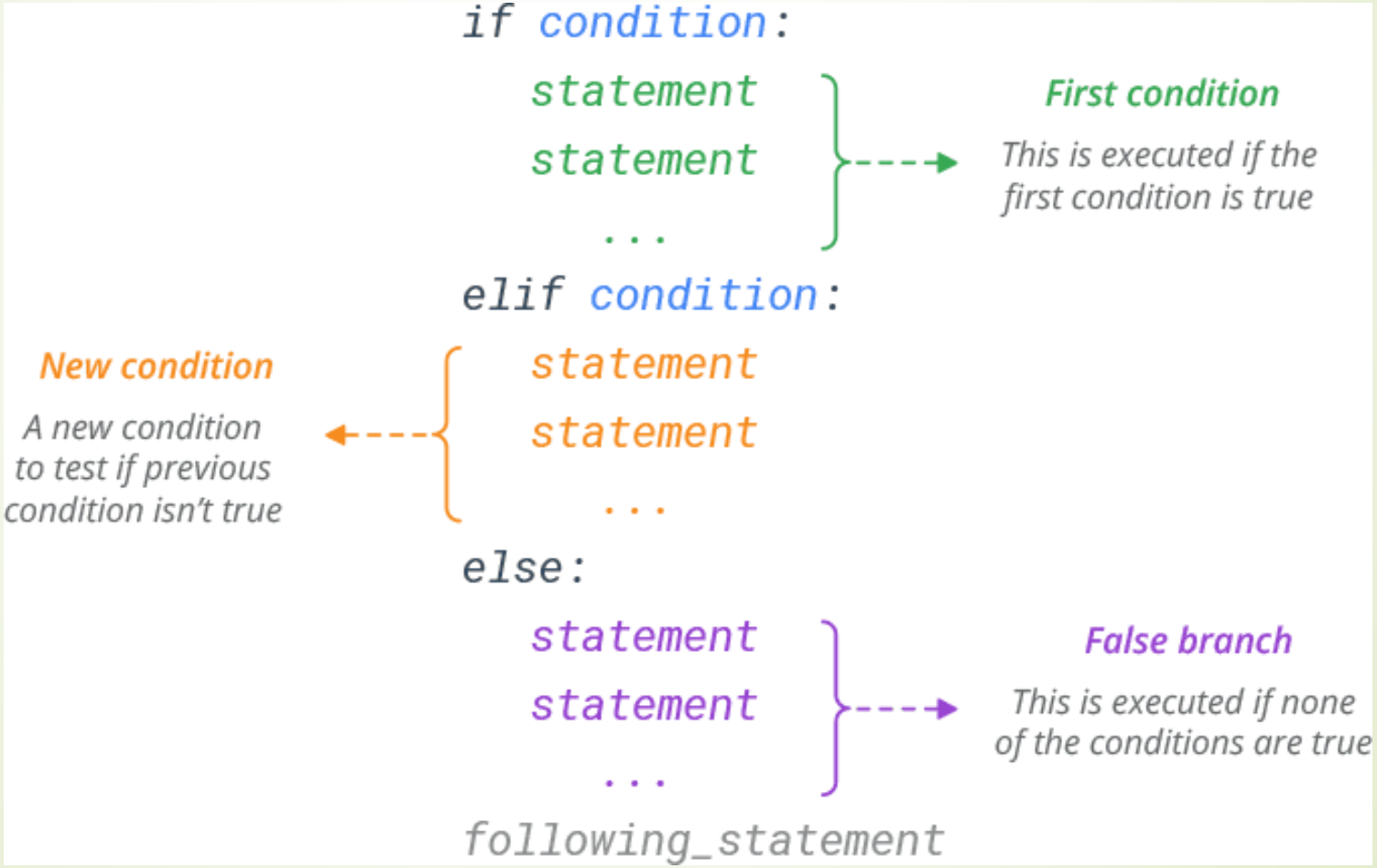
If you run the code and enter a number, the script will print a different message based on whether the number is greater or less/equal to 5.

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

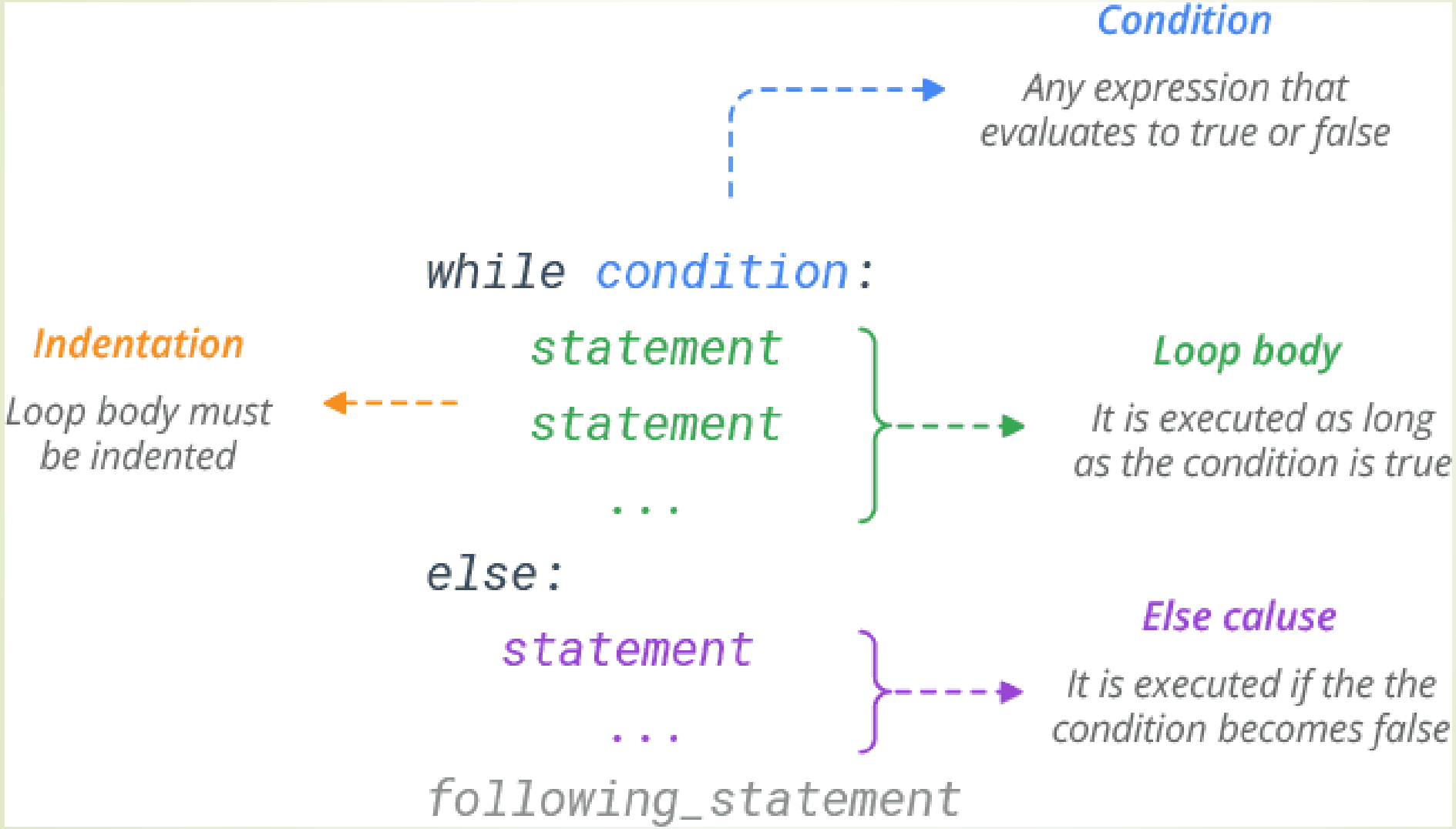
while your tea is too hot, add a chip of ice.

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

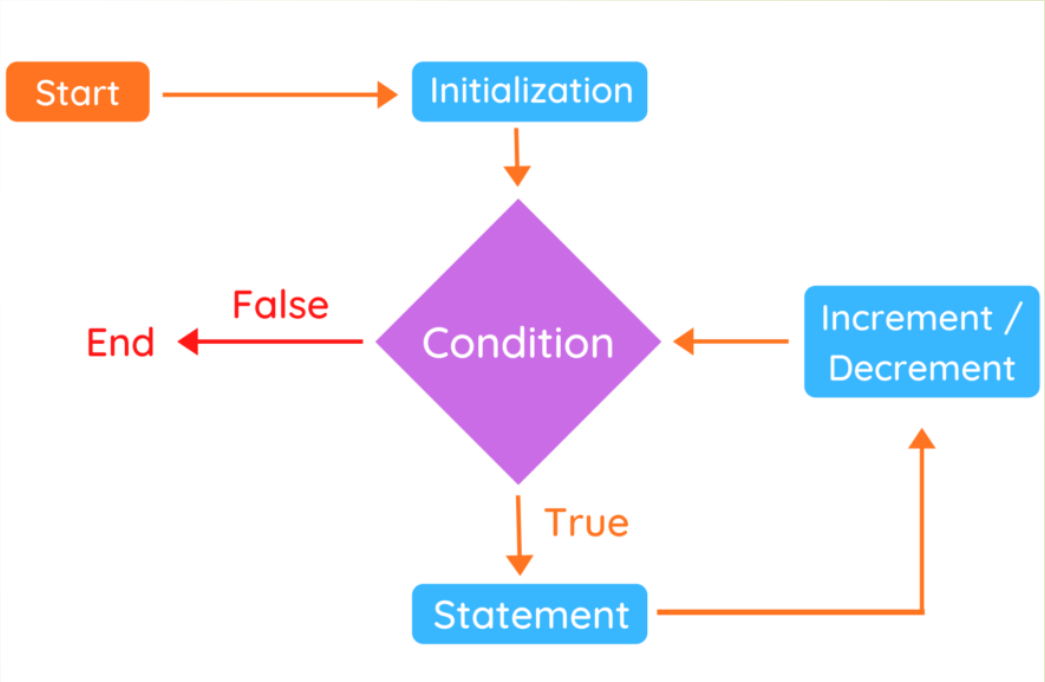


While loop

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!
```