Lecture 9 – Iterations

Iteration repeats the execution of a sequence of code. Iteration is useful for solving many programming problems.

In computer programming **a *loop* *or* iteration is statement or block of statements that is executed repeatedly**.

Python has two kinds of loops, a **`for`** loop and a **`while`** loop.

- First, introduce the **`for`** loop and illustrate its use for a variety of tasks.
- Then, introduce the **`while`** loop.
- **Compare** the two kinds of loops and discuss when to use one or the other.

`for` loops

The **for** statement iterates over a range of values.

These values can be a numeric range, or, as we shall, elements of a data structure like a string, list, or tuple. T

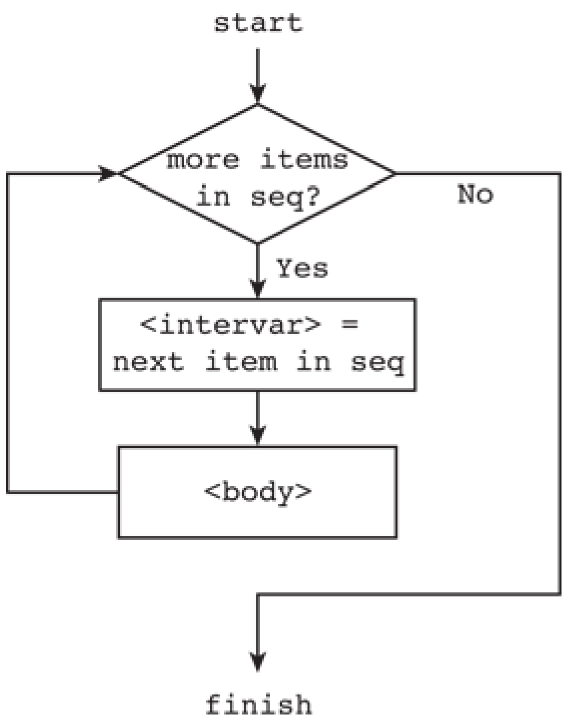
The general form of a **`for`** loop in Python is

```
for <itervar> in <sequence>:
    <body>
```

where `<intervar>` is a variable, `<sequence>` is a sequence such as list or string or array, and `<body>` is a series of Python commands to be executed repeatedly for each element in the `<sequence>`.

The `<body>` is indented from the rest of the text, which defines the extent of the loop.

Flowchart for `for`-loop.



The figure above shows the flowchart for a `for` loop.

- It starts with an implicit conditional asking if there are any more elements in the sequence.
- If there are, it sets the iteration variable equal to the next element in the sequence and then executes the body—the indented text—using that value of the iteration variable.
- It then returns to the beginning to see if there are more elements in the sequence and continues the loop until there is none remaining.

Example

```
for dogname in ["Max", "Molly", "Buster", "Maggie", "Lucy"]:
    print(dogname)
    print("    Arf, arf!")
print("All done.")
```

Sample Run

```
Max
    Arf, arf!
Molly
    Arf, arf!
Buster
    Arf, arf!
Maggie
    Arf, arf!
Lucy
    Arf, arf!
All done.
```

The `for` loop works as follows:

- The *iteration variable* or *loop index* `dogname` is set equal to the first element in the list, `"Max"`, Then the two lines in the indented body are executed.
- Then `dogname` is set equal to second element in the list, `"Molly"`, and the two lines in the indented body are executed.
- The loop cycles through all the elements of the list.
- Finally , it moves on to the code that follows the `for` loop and prints `All done.`

When indenting a block of code in a Python `for` loop, **it is critical that every line be indented by the same amount**. Using the **<tab>** key causes the Code Editor to indent 4 spaces. Any amount of indentation works, as long as it is the same for all lines in a `for` loop.

Example

```
for num in range(1, 10, 2):
     print(num)
```

Sample Run

```
1
3
5
7
9
```

Example

```
for i in range(20,25):
    print(i)
```

Sample Run

```
20
21
22
23
24
```

Example

```
for i in range(50,0,-10):
    print(i)
```

Sample Run

```
50
40
30
20
10
```

Example

```
# loop will not get executed Because 20
# is not < 10

for i in range(20,10):
    print(i)
print("\n\nOut of the loop")
```

Sample Run

```
Out of the loop

>>>
```

Example

```
numbers = [0, 254, 2, -1, 3 , 0 , -9 , 8 ]
for num in numbers:
  if (num < 0):
    print(num , "\tNegative number is detected!")
  elif (num == 0):
    print(num , "\tZero number is detected!")
  else: print(num , "\tPositive number is detected!")
```

Sample Run

```
0    Zero number is detected!
254  Positive number is detected!
2    Positive number is detected!
-1   Negative number is detected!
3    Positive number is detected!
0    Zero number is detected!
-9   Negative number is detected!
8    Positive number is detected!
>>>
```

Example

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```
for i in reversed(range(1, 10, 2)):
    print(i)
```

Sample Run

```
9
7
5
3
1
```

Example

```
sammy = 'Sammy'
for letter in sammy:
     print(letter)
```

Sample Run

```
S
a
m
m
y
```

You can also use a `for` loop to construct a list from scratch:

Example

```
integers = []

for i in range(10):
   integers.append(i)

print(integers)
```

Sample Run

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>>
```

Iterating Through Dictionary

When iterating through a dictionary, it’s important to keep the key : value structure in mind to ensure that you are calling the correct element of the dictionary.

Example

```
sammy_shark = {'name': 'Sammy', 'animal': 'shark',
'color': 'blue', 'location': 'ocean'}
```

```
for key in sammy_shark:
   print(key + ': ' + sammy_shark[key])
```

Sample Run

```
name: Sammy
animal: shark
color: blue
location: ocean
```

Accumulators

As another application of Python's `for` loop , suppose you want to calculate the sum of all the odd numbers between 1 and 100.

- Before writing a computer program to do this, think about how it is done by hand.
- Start tart by adding 1+3=4.
- Keep track of sum
- Then take the sum 4 and add the next odd integer, 5, to get 4+5=9;
- Then 9+7=16, then 16+9=25, and so forth.
- Repeated additions is done, starting with 1+3 until the last number 99 is reached.

Must keep track of the running sum with the variable **`s`**, which is called the ***accumulator*.**

- Initially `s=0`.
- Then **add** the first number, **1, to `s`** and **`s`** becomes **1**.
- Then **add** then next number, **3 to `s`** and **`s`** becomes **4**.
- **Continue** doing this over and over again using a `for` loop while the variable **`s` accumulates** the running sum until the final number is reached.

Example

```
s = 0
for i in range(1, 100, 2):
    s = s+i
print(s)
```

Sample run

```
2500
```

The **range** function defines the list [1, 3, 5, ..., 97, 99].

The for loop successively adds each number in the list to the running sum until it reaches the last element in the list and the sum is complete.

Once the for loop finishes, the program exits the loop and the final value of s, which is the sum of the odd numbers from 1 to 99, is printed out.

Nested For Loops

Loops can be nested in Python, as they can with other programming languages.

A nested loop is a loop that occurs within another loop, structurally similar to nested if statements.

```
# Outer loop

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

    # Nested loop
      [do something]
```

The program first encounters the outer loop, executing its first iteration.

This first iteration triggers the inner, nested loop, which then runs to completion.

Then the program returns back to the top of the outer loop, completing the second iteration and again triggering the nested loop.

Again, the nested loop runs to completion, and the program returns back to the top of the outer loop until the sequence is complete or a break or other statement disrupts the process.

Example

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

for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
```

Sample Run

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

Example

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)

print("\nOutside the nested loop ")
```

Sample Run

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry


Outside the nested loop
```

Example

```
names=['Foster','Emma']

cars=['Jeep','Lexus','Hyundai' ]

numbers=[1,2]

for name in names:

  for car in cars:

    for number in numbers:

      print(name+" has "+str(number)+" "+car)
```

Sample Run

```
Foster has 1 Jeep

Foster has 2 Jeep

Foster has 1 Lexus

Foster has 2 Lexus

Foster has 1 Hyundai

Foster has 2 Hyundai

Emma has 1 Jeep

Emma has 2 Jeep

Emma has 1 Lexus

Emma has 2 Lexus

Emma has 1 Hyundai

Emma has 2 Hyundai
```

`while` loops

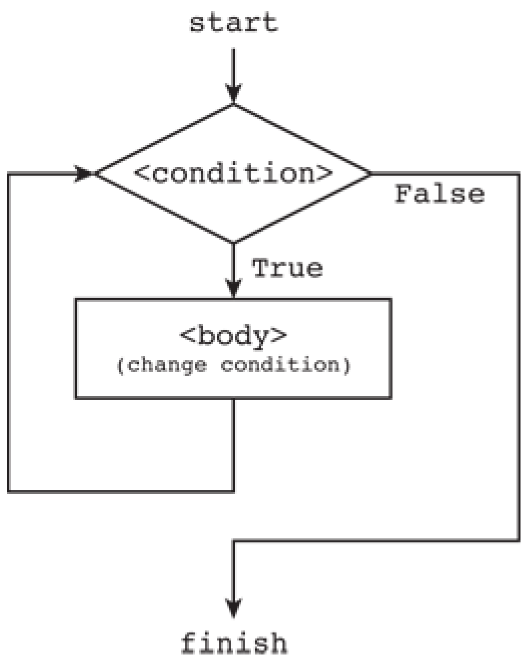
The general **form of a `while`** loop in Python is

```
while [a condition is True]:
 [do something  /  body ]
```

- `<condition>` is a statement that can be either `True` or `False`
- `<body>` is a series of Python commands that is executed repeatedly until `<condition>` becomes false.

- This means that somewhere in `<body>`, the truth value of <condition> must be **changed** so that it becomes false after a finite number of iterations.

Flowchart for `while` loop.



Example

```
number = 5
sum = 0
i = 0

while ( i<number):
        sum = sum + i
        i = i+1

print(sum)
```

Sample Run

```
10

>>>
```

Example

Write a small program that executes a while loop. In this program, ask the user to input a password.   There are two possible outcomes:

- If the password is correct, the while loop will exit.
- If the password is not correct, the while loop will continue to execute.

Solution

```
password = ''
print('Enter A Password   ')
password = input()
while password != 'password':
    print('Incorrect Password , Try Again ')
    password = input()
print("Welcome ")
```

Sample Run

```
Enter A Password
aaa
Incorrect Password , Try Again
yyyyyyy
Incorrect Password , Try Again
password
Welcome
```

Nested While loop

Example

```
i = 1 # Outer iterator

while i <= 5:
    j = 1 # Inner iterator
    while j <= 5:
        print( i*j , end='  ' ) # Prevent python from printing new line
        j += 1
    print() # To print new line
    i += 1
```

Sample Run

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12  15
4  8  12  16  20
5  10  15  20  25
>>>
```

while and for loop together

Example

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

while x != 0:
    for y in range (1, x):
        print (y)
        #y+=1
    x = int(input('Enter a number: '))

print("\nNested loop is terminated")
```

Sample Run

```
Enter a number: 4
1
2
3
Enter a number: 6
1
2
3
4
5
Enter a number: 0

Nested loop is terminated
>>>
```

break Statements

There is a shortcut to getting the program execution **to break out of a while loop's clause early**.

If the execution reaches a break statement, it immediately exits the while loop's clause.

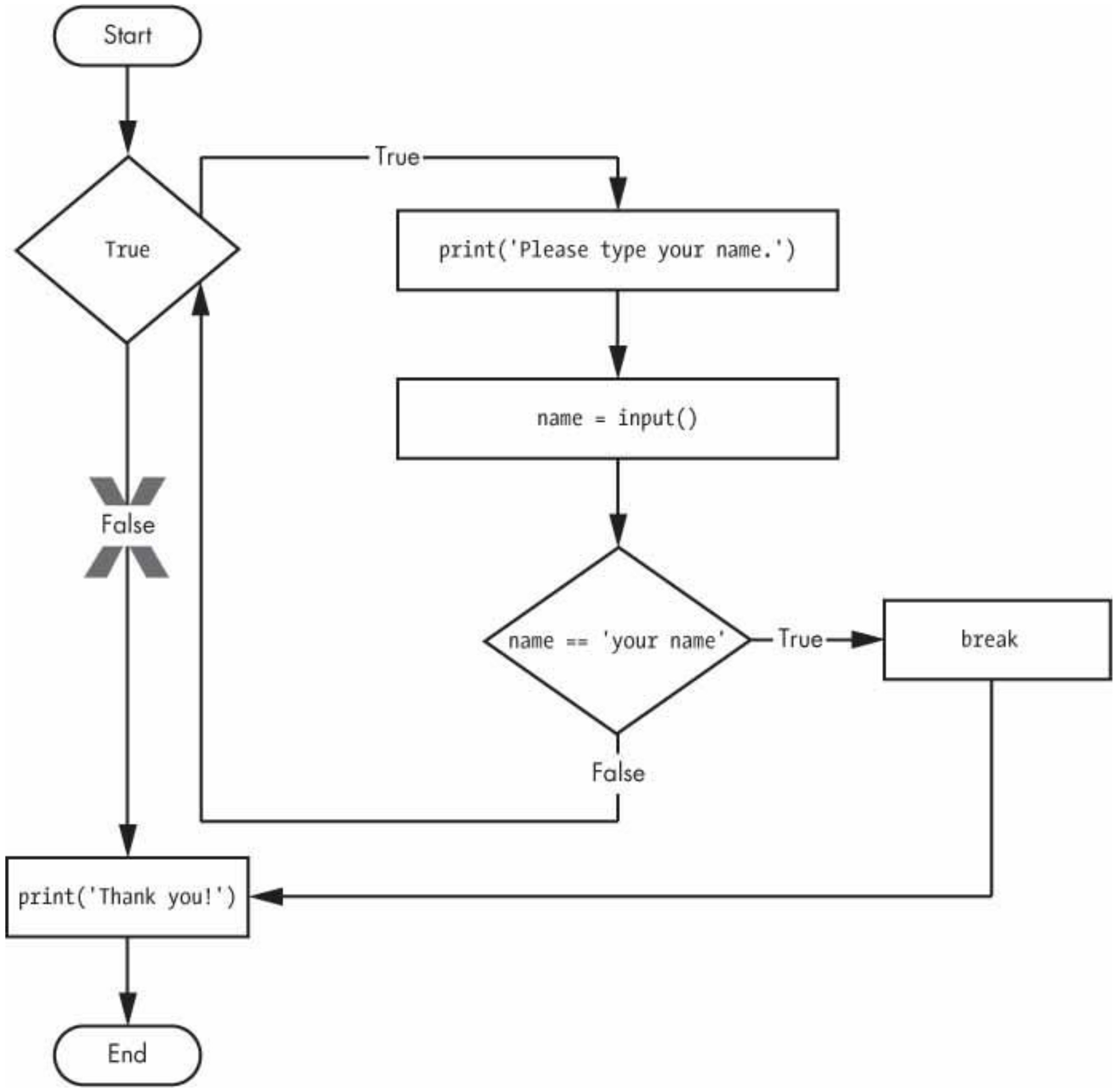
In code, a break statement simply contains the break keyword.

Example

```
while True:
    print('Please type your name.')
    name = input()
    if name == 'X':
      break
print('Thank you!')
```

Sample Run

```
Please type your name.
Bill
Please type your name.
George
Please type your name.
your name
Thank you!
>>>
```


Start
True
True
print('Please type your name.')
name = input()
name == 'your name'
True
break
False
False
print('Thank you!')
End

Continue Statements

Like break statements, **continue** statements are used inside loops.

When the program execution reaches a continue statement, **the program execution immediately jumps back to the start of the loop** and reevaluates the loop's condition.

Example

```
for each in range(10):
    if each % 2:     # executed as not equal
        continue
    if each > 7:
        break
    print(each)
```

Sample Run

```
0
2
4
6
>>>
```

Show the output of the following program and convert the for statement to while statement

```
for each in range(10):
    if ( each % 2  == 0 ) :
        continue
    if each > 7:
        break
    print(each)

print("\nOut of the Loop ")
```

Sample Run

```
1
3
5
7

Out of the Loop
>>
```

From for to while

```
each = 0
while each < 10:

    if ( each % 2 == 0 ) :
        each = each + 1
        continue
    if each > 7 :
        break
    print (each)
    each = each + 1

print("\nOut of the Loop ")
```

Using else statement with for loops

You can also combine else statement with for loop. The else block will be executed immediately after for block finishes execution.

```
# Python program to illustrate
# combining else with for

index=0
list = ["Hi", "Bye", "Python"]
for index in range(len(list)):
    print( list[index] )
else:
    print ("Inside Else Block")

print ("Outside The for loop")
```

Example

Sample Run

```
Hi
Bye
Python
Inside Else Block
Outside The for loop
>>>
```

Using else statement with for loops and break

```
# Python program to illustrate
# combining else with for and break

index=0
list = ["Hi", "Why" , "Good" , "Bye", "Python"]
for index in range(len(list)):
    if ( index == 3 ) :
        break
    else:
        print( list[index] )
else:
    print ("Inside Else Block")

print ("Outside The for loop")
```

Sample Run

```
Hi
Why
Good
Outside The for loop
>>>
```

Using else statement with while loops

- As discussed above, while loop executes the block until a condition is satisfied.
- When the condition becomes false, the statement immediately after the loop is executed.
- The **else** clause is only executed when the **while** condition becomes **false**.
- If you break out of the loop, or if an exception is raised, the else won't be executed.

# Python program to illustrate combining else with while

```
count = 0
while (count < 5):
    count = count + 1
    print("Hello")
else:
    print("\n\nIn Else Block")

print("\n\nOut of while")
```

Sample Run

```
Hello
Hello
Hello
Hello
Hello

In Else Block
Out of while

>>>
```

Using else statement with while loops and break

Example

```
#Python program to illustrate
# combining else with while and break
count = 0
while (count < 5):
    count = count + 1
    if ( count == 3 ):
         break
    else:
        print("Hello")
else:
    print("In Else Block")

print("Out of while")
```

Sample Run

```
Hello
Hello
Out of while
>>>
```

Exercise :

- What is the output of the following program
- Convert The following Python code from for to while.

```
for x in range(6,0,-1):
    for y in range(x , 9 , 1 ):
        print("*",end=' ')
    print()
```

Exercise :

- What is the output of the following program
- Convert The following Python code from for to while.

```
for x in range(10):
  if x > 5:
    x += 2
    continue
  x = x + 1
  print ("Still in the loop. ",x)
  if x == 8:
    break
print ("Outside of the loop.")
```

Exercise :

- What is the output of the following program
- Convert The following Python code from while to for.

```
number=0
while number < 5 :
    if number%2 == 0:
        print("The number "+str(number)+" is even")
    else:
        print("The number "+str(number)+" is odd")

    number = number+1
```