



Basics of Programming through Python

Repetition Structure

Introduction to
programming COMP102

Term 3-2022-2023



Iteration /Loops





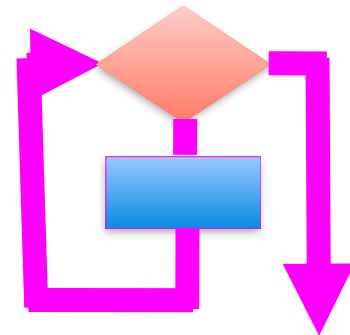
Lesson Outcomes

- Use loops to solve common problems, such as, finding sum, finding max, etc.
- Write different types of loops, such as for, while and do-while.
- Select the correct type of loop based on a given problem.



ITERATION **OR** LOOPING

- Loops can execute a block of code number of times until a certain condition is met.
- The iteration statement allows instructions to be executed until a certain condition is to be fulfilled.
- The iteration statements are also called as loops or Looping statements



- Python provides two kinds of loops & they are:

while loop

for loop

The **while** statement

- A while loop is a programming concept that, when it's implemented, executes a piece of code repeatedly while a given condition still holds **true**. The above definition also highlights the three components that you need to construct the while loop in Python:
 - ✓ The **while** keyword;
 - ✓ A condition that is evaluated to either **True** or **False**; And
 - ✓ A block of code that you want to execute repeatedly
- 1. If the condition is **false**, exit the while statement and continue execution at the next statement.
- 2. If the condition is **true**, execute the body and then go

While Loop

- The syntax for a while loop in Python is as follows:

while condition:
body

Where, loop body contain the single statement or set of statements (compound statement) or an empty statement.

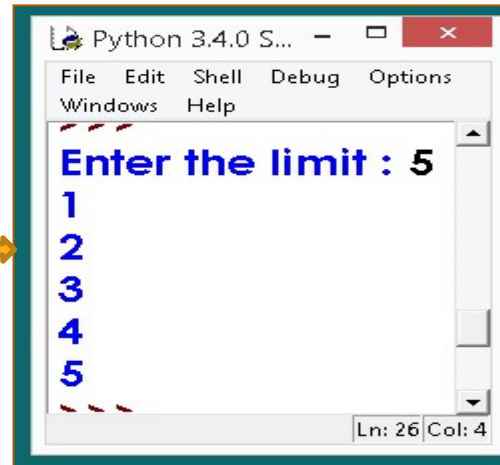
General way to write a while loop

Initialize your counter	A variable is initialized outside of the loop
while <condition>:	While loop executes while the condition is True
Statements	Executes the code within the block of code
counter update	Counter is updated inside the loop
Instructions	More instructions after the while loop(optional)

Example 1: print natural numbers

```
i=1  
n=int(input("Enter the limit : "))  
while(i<=n):  
    print(i)  
    i+=1
```

OUTPUT



The screenshot shows a Python 3.4.0 Shell window with a menu bar (File, Edit, Shell, Debug, Options, Windows, Help). The main text area displays the prompt "Enter the limit : 5" in blue. Below it, the numbers 1, 2, 3, 4, and 5 are printed on separate lines, also in blue. The status bar at the bottom right indicates "Ln: 26 Col: 4".

Example 2: Calculating Sum of Natural Numbers

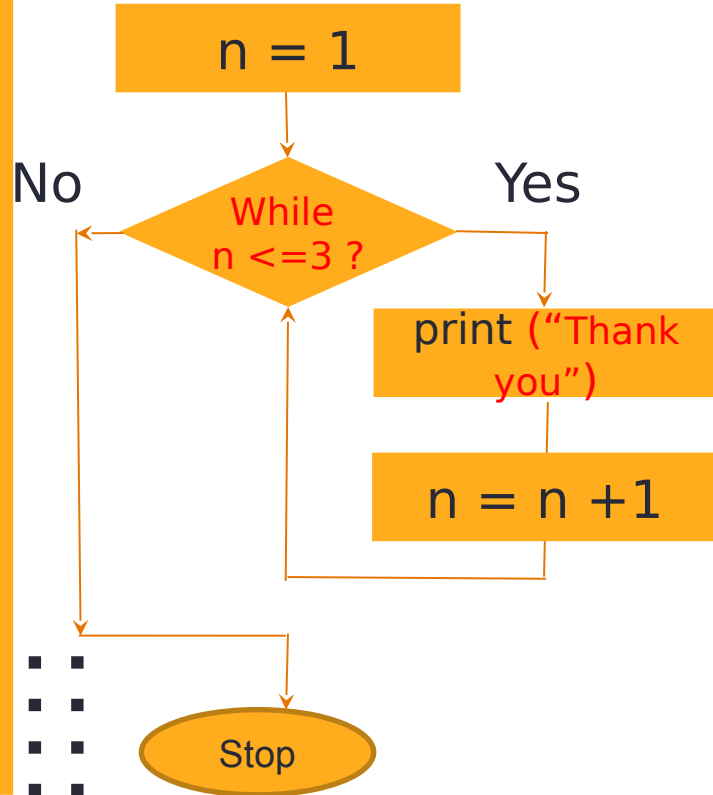
```
sum1 = 0
count = 1
while (count < 10):
    sum1 = sum1 + count
    count = count + 1
print (count) # should be 10
print (sum1) # should be 45
```

Updating variables

- Updating a variable by adding 1 is called an *increment*;
- subtracting 1 is called a *decrement*

Note: You can add or subtract any value not just 1

The while statement : Example (Displaying "Thank you" 3 times)



Program

```
# initialization
n = 1
# Condition of the while loop

while n <= 3:
    print ("Thank you")
    # Increment the value of
    the variable "n by 1"
    n = n+1
```

Output

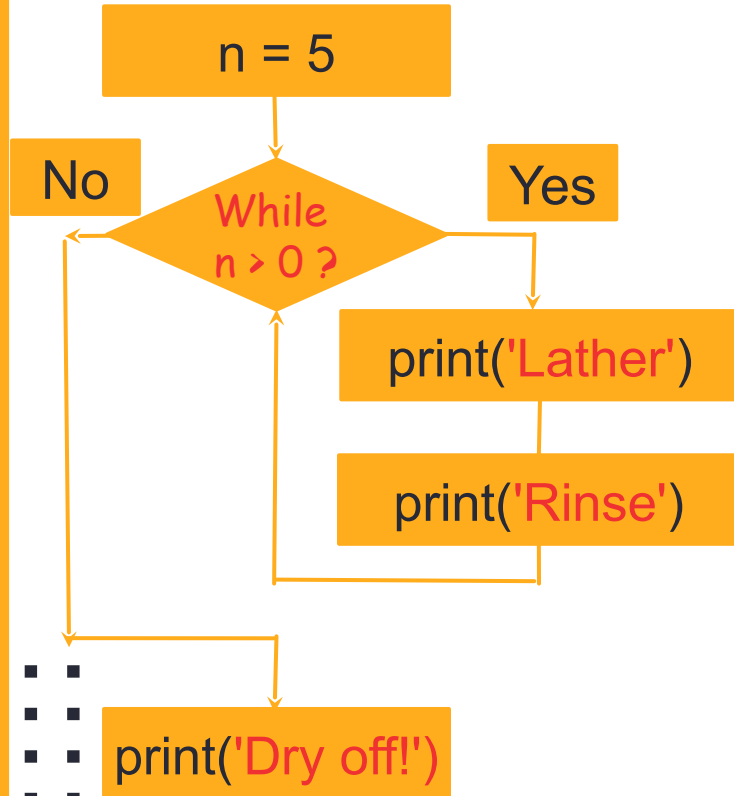
Thank you

Thank you

Thank you

Loops (repeated steps) have **iteration variables** that change each time through a loop. Often these **iteration variables** go through a sequence of numbers.

An Infinite Loop (no counter update)



■ Program

```
n = 5
while n > 0 :
    print ("Lather")
    print ("Rinse")
print ("Dry off")
```

■ Output

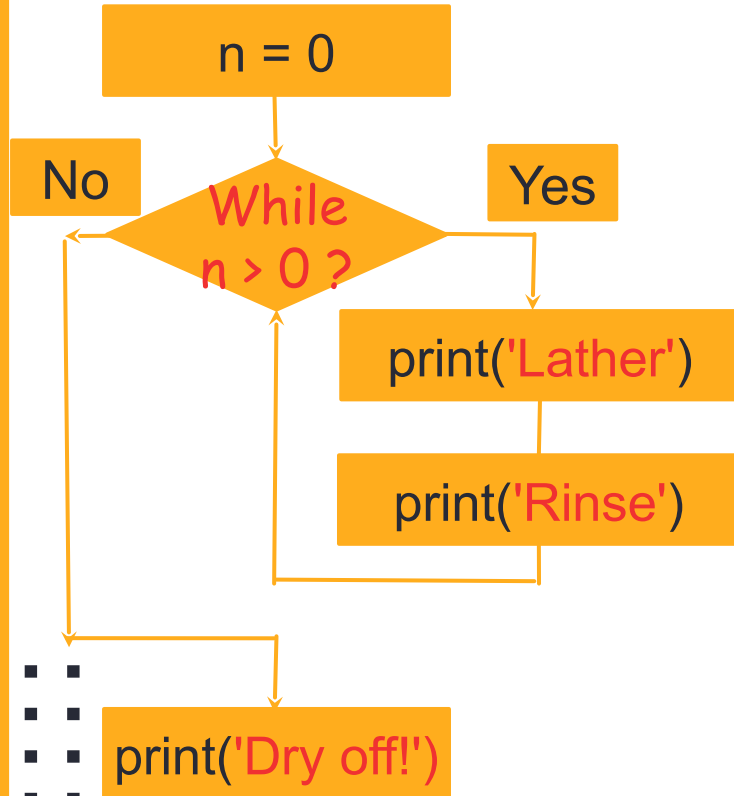
"Lather"

"Rinse"

Will be infinitely displayed
and "Dry off" will
never be displayed

This loop is obviously an **infinite** loop because the **logical expression** on the while statement is simply the logical constant **True**

Another loop



■ Program

```
n = 0
while n > 0 :
    print ("Lather")
    print ("Rinse")
print ("Dry off")
```

■ Output

Dry off

This program will not display “Lather” and “Rinse” because the logical expression on the while statement is False from the first iteration

Breaking Out of a Loop : The infinite loop example 1

- The **break** statement ends the current loop and jumps to the statement immediately following the loop.
- It is like a loop test that can happen anywhere in the body of the loop.

■ Program

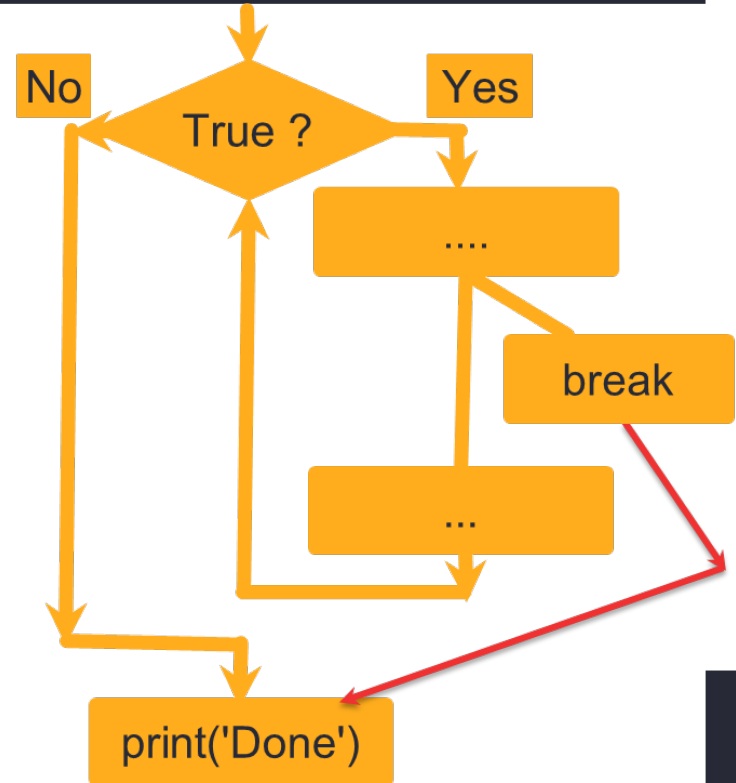
```
n=0
while n >= 0 :
    stop = input("enter
any string")
    if stop == "done":
        break
    print ("Lather")
    print ("Rinse")
print ("Dry off")
```

■ Output

Each time through, it prompts the user with an angle bracket. If the user types done, the **break** statement exits the loop and print "Dry off". Otherwise, the program displays "Lather" "Rinse" and goes back to the the loop

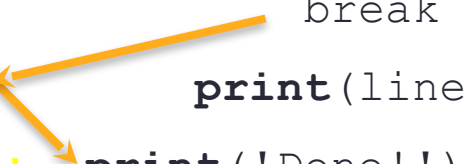
Breaking Out of a Loop : The infinite loop example 2

```
• while True:
    line = input('enter
    a string ')
    if line == 'done' :
        break
    print(line)
• print('Done!')
```



Breaking Out of a Loop : The infinite loop example 2

```
• while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
• print('Done!')
```



- The loop condition is **True**, which is always true, so the loop runs repeatedly until it hits the break statement. Here's a sample run:

> hello there

hello there

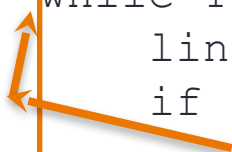
> finished

finished

> done

Finishing iterations with continue

The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration



```
while True:
    line = input(">")
    if line[0] == "#" :
        continue
    if line == 'done' :
        break
    print(line)
print("Done!")
```

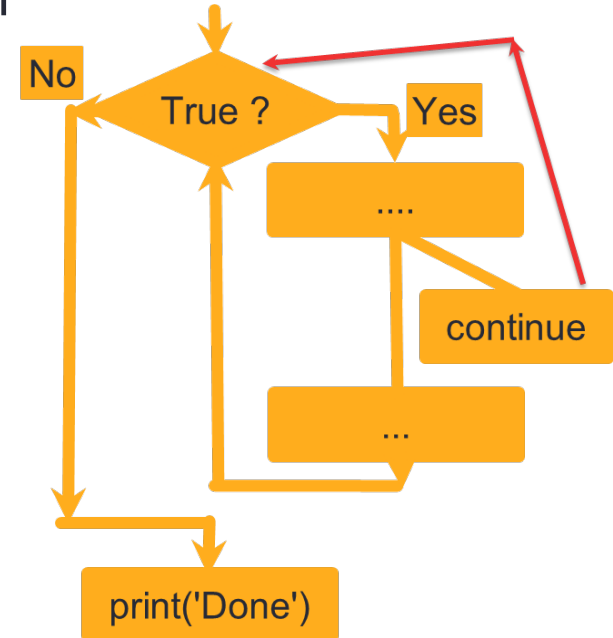
■ Output

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Finishing iterations with continue

The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input(">")
    if line[0] == "#" :
        continue
    if line == 'done' :
        break
    print(line)
print("Done!")
```



Difference Between break and continue

BREAK	CONTINUE
It terminates the execution of remaining iteration of the loop.	It terminates only the current iteration of the loop.
'break' resumes the control of the program to the end of loop enclosing that 'break'.	'continue' resumes the control of the program to the next iteration of that loop enclosing 'continue'.
It causes early termination of loop.	It causes early execution of the next iteration.
'break' stops the continuation of loop.	'continue' do not stops the continuation of loop, it only stops the current iteration.

pass statement

- The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
- The pass statement is a **null operation**;
■ nothing happens when it executes.

■ Example:

```
1 for i in range(0,10):  
2     pass  
3 print("Good bye!")
```

Output: Good bye!

for loop

- Python's **for-loop** syntax is a more convenient alternative to a while loop when iterating through a series of elements. The for-loop syntax can be used on any type of iterable structure, such as a list, tuple, str, set, dict, or file.
- We can write a loop to run the loop once for each of the items in a set using the Python for construct.

Definite loops using **for**

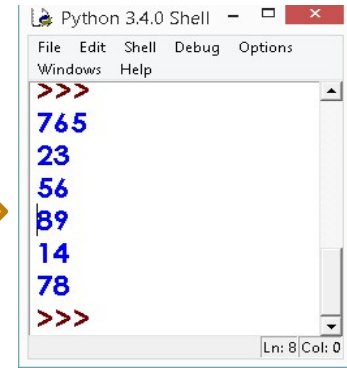
- These loops are called “**definite loops**” because they execute an exact number of times.
- Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the sequence or set.
- We say that “**definite loops iterate through the members of a set**”
- **Syntax:**

```
for counter in range():  
    statements
```

Example 1: print items of list

```
numbers=[765,23,56,89,14,78]  
for i in numbers:  
    print(i)
```

OUTPUT



A screenshot of a Python 3.4.0 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The main text area shows the prompt '>>>' followed by the numbers 765, 23, 56, 89, 14, and 78, each on a new line. The prompt '>>>' appears again at the bottom. The status bar at the bottom right shows 'Ln: 8|Col: 0'.

```
>>>  
765  
23  
56  
89  
14  
78  
>>>
```


Range() function

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

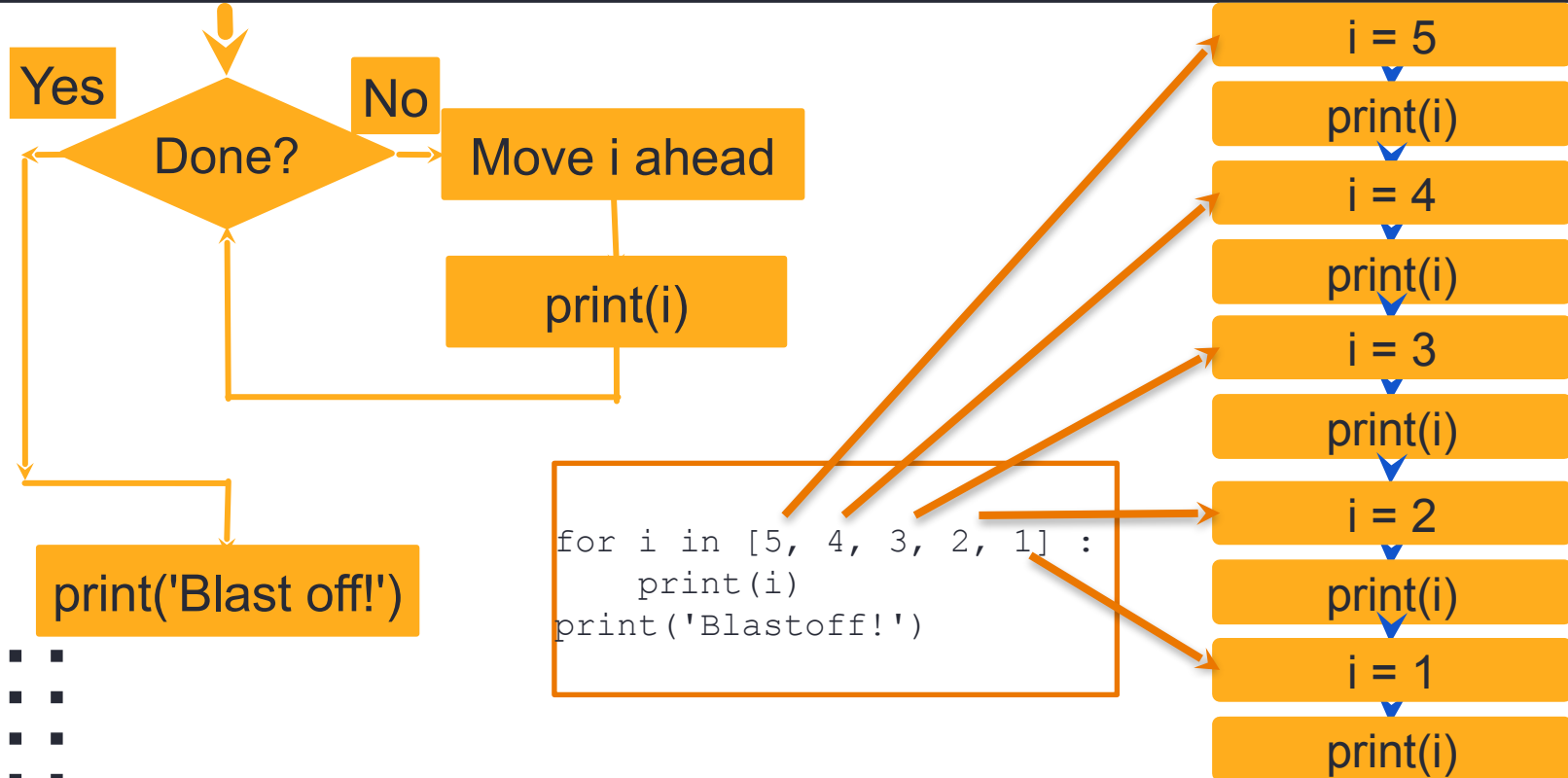
range(start, stop, step)

```
for n in range(3,6):  
    print(n)
```

Output:

3
4
5

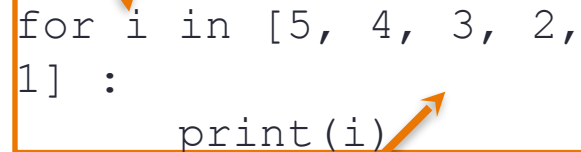
Definite loops using for : Example 1



Definite loops using for : Example 1

- The iteration variable “iterates” through the sequence (ordered set)
- The block (body) of code is executed **once** for each value in the sequence
- The iteration variable moves through all of the values in the sequence

Iteration variable



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

Five-element
sequence

Definite loops using for : Example 2

■ Program

```
friends = ["Ahmad", "Ali", "Saad"]  
for friend in friends :  
    print("Good Morning:", friend)  
print('Done!')
```

In Python terms, the variable friends is a list of three strings and the for loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

Good Morning: Ahmad
Good Morning: Ali
Good Morning: Saad
Done!

For loop Example 3

■ Program 1

```
for i in "great":  
    print(i)
```


■ Output 1

g
r
e
a
t

For loop and Range function

■ Program 1

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



To print in horizontal not vertical line
If not used (end="") will print vertically

■ Program 2

```
for b in range(1, 5):  
    print(b, end="")
```

■ Output 1


0 1 2 3 4 5 6 7 8 9

■ Output 2

1 2 3 4

■ Program 3

```
for c in range(2, 10, 2):  
    print(c, end="")
```



■ Because its start from 0 using index

Note : can be in reverse using -

For loop and Range function

■ Program 1

```
for j in range(4):  
    print("# ")
```

■ Program 2

```
for j in range(4):  
    print("# ",end="")
```

■ Output 1

```
#  
#  
#  
#
```

■ Output 2

```
# # # #
```

For loop and Range function

■ Program 3

```
for j in range(4):  
    print("# ",end="")
```

```
Print()
```

```
for j in range(4):  
    print(#,end="")
```

```
Print()
```

■ Output 3

```
# # # #
```

```
# # # #
```


For loop and **Range** function

■ Program 4

```
for i in range(3):  
    for j in range(4):  
        print("# ",end="")  
  
Print()
```

■ Output

```
# # # #  
# # # #  
# # # #
```

The program will execute 12 times to
execute Loop

else statement in loop

- **else** can be used in **for** and **while** loops. The else body will be executed when the loop's conditional expression evaluates to **false**

```
1 ▾ for i in range(0,5):  
2     print(i)  
3 ▾ else:  
4     print("else of for loop is executed")
```



```
0  
1  
2  
3  
4  
else of for loop is executed  
> |
```

Loop patterns : Looping Through a Set

■ Program

```
print('Before')
for thing in [9, 41, 12, 3, 74, 15] :
    print(thing)
print('After')
```

■ Output

Before

9
41
12
3
74
15

After

Loop patterns : Counting the number of items in a list

- We set the variable count to zero before the loop starts,
- we write a for loop to run through the list of numbers using an iteration variable
- In the body of the loop, we add 1 to the current value of count for each of the values in the list.
- Once the loop completes, the value of count is the total number of items. We construct the loop so that we have what we want when the loop finishes.

Loop patterns : Counting the number of items in a list

■ Program

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

■ Output

Count: 6

Loop patterns : computing the total of a set of numbers

- In this loop we do use the iteration variable. Instead of simply adding one to the count as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration.
- So before the loop starts total is zero because we have not yet seen any values, during the loop total is the running total, and at the end of the loop total is the overall total of all the values in the list.
- As the loop executes, total accumulates the sum of the elements; the items in the list respectively.

Loop patterns : computing the total of a set of numbers

■ Program

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

■ Output

Total: 154

Loop patterns : find the largest -Maximum value in a list or sequence

■ Program

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

■ Output

Before: None

Loop: 3 3

Loop: 41 41

Loop: 12 41

Loop: 9 41

Loop: 74 74

Loop: 15 74

Largest: 74

Loop patterns : find the largest value in a list or sequence

- Before the loop, we set largest to the constant None. None is a special constant value which we can store in a variable to mark the variable as “empty”. Before the loop starts, the largest value we have seen so far is None since we have not yet seen any values.
- While the loop is executing, if largest is None then we take the first value we see as the largest so far. You can see in the first iteration when the value of itervar is 3, since largest is None, we immediately set largest to be 3.
- After the first iteration, largest is no longer None, so the second part of the compound logical expression that checks itervar > largest triggers only when we see a value that is larger than the “largest so far”. When we see a new “even larger” value we take that new value for largest. You can see in the program output that largest progresses from 3 to 41 to 74. At the end of the loop, we have scanned all of the values and the variable largest now does contain the largest value in the list.

Loop patterns : find the smallest-Minimum value in a list or sequence

■ Program

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

■ Output

```
Before: None
Loop: 3 3
Loop: 41 3
Loop: 12 3
Loop: 9 3
Loop: 74 3
Loop: 15 3
Smallest: 3
```

Again, smallest is the “smallest so far” before, during, and after the loop executes. When the loop has completed, smallest contains the minimum value in



Exercises



Exercise 1

Write a python program to print the square of all numbers from 0 to 10.

[0,1,2,3,4,5,6,7,8,9,10]

```
for x in range(11):  
    #range(0,11)  
    c=x**2  
    print(c)    #  
print(c,end=" ")  
# can merge line2 & 3  
print(x**2)
```

```
i=0  
while i<=10:  
    print("square of", i,"is  
equal to", i**2)  
    i=i+1  
    #i+=1  
print("*****end  
of the  
program*****")
```

Exercise 2

Write a python program to find the sum of all even numbers from 0 to 10

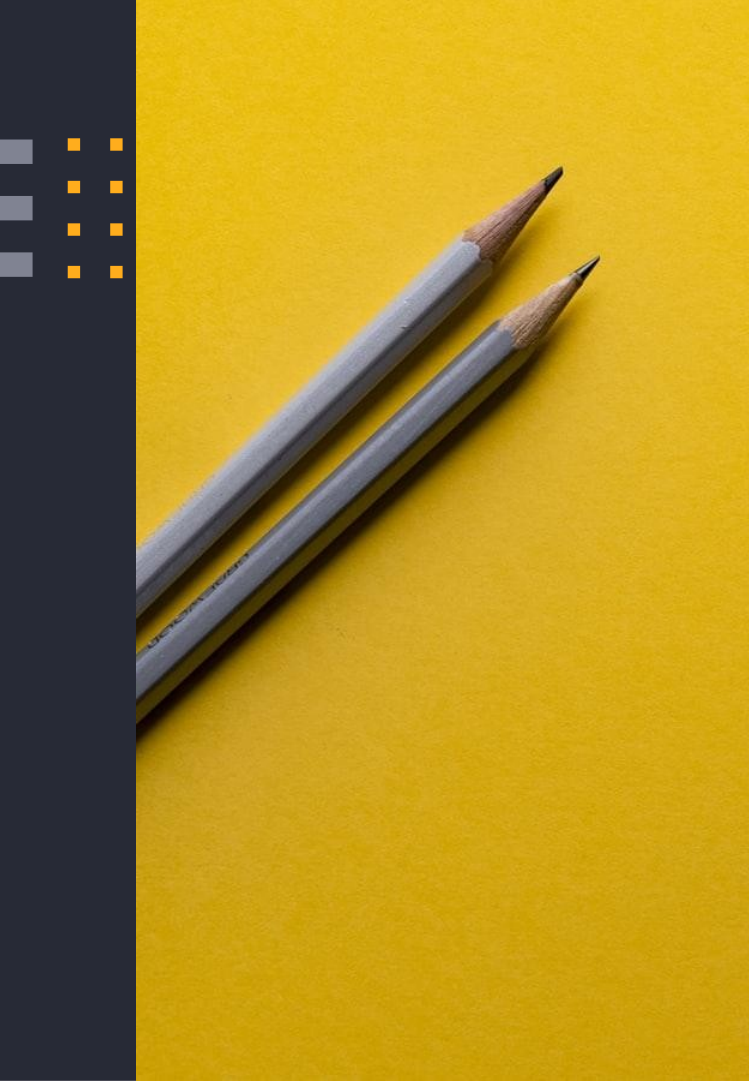
```
s=0    # sum
for x in range(11):
    if x%2==0:
        s=x+s
print(s)
```

```
s=0
x=0
#counter
while x<=10:
    if x%2==0:
        s=x+s
        x=x+1
#x+=1
print(s)
```

Exercise 3

- Write a python program to read three numbers (a,b,c) and check how many numbers between 'a' and 'b' are divisible by 'c'. Note: use the highest value for a .
- For example: a=10,b=2,C=5
- Output: Between 2 and 10 there is 2 numbers divisible by 5 (5,10)

```
a,b,c=eval(input("enter the three numbers:"))
x=0
if a>b:
    for i in range(b,a+1):
        if i%c==0:
            print(i)
            x=x+1
print("The total numbers able to divide is:",x)
```



Thanks!

