3.THE FOR LOOP (for Collections)

The **for** loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The **for** statement works on ***strings, lists, tuples, and other built-in iterables***, as well as ***new user-defined objects***.

```
for entity in collection:       # Assign object items to target
    statements                  # Repeated loop body: use target
else:                           # Optional else part
    statements                  # If we didn't hit a 'break'
```

When Python runs a **for** loop, it assigns the items in the iterable object to the target one by one and executes the loop body for each. The loop body typically uses the assignment target (entity) to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

The name used as the assignment target in a **for** header line is usually a (possibly new) variable in the scope where the for statement is coded. There's not much unique about this name; it can even be changed inside the loop's body, but it will automatically be set to the next item in the sequence when control returns to the top of the loop again.

After the loop this variable normally still refers to the last item visited, which is the last item in the sequence unless the loop exits with a break statement.

The **for** statement also supports an optional **else** block, which works exactly as it does in a **while** loop—it's executed if the loop exits without running into a **break** statement (i.e., if all items in the sequence have been visited). The **break** and **continue** statements introduced earlier also work the same in a **for** loop as they do in a **while**. The **for** loop's complete format can be described this way:

```
for item in object:         # Assign object items to target
    statements
    if test: break          # Exit loop now, skip else
    if test: continue       # Go to top of loop now
else:
    statements
```

As mentioned earlier, a **for** loop can step across any kind of sequence object.

CONSTRUCTS - B

In our first example, for instance, we'll assign the name x to each of the three items in a list in turn, from left to right, and the print statement will be executed for each. Inside the print statement (the loop body), the name x refers to the current item in the list:

```
>>> for x in ["spam", "eggs", "ham"]:
        print(x, end=' ')

spam eggs ham

>>> for x in ["spam", "eggs", "ham"]:
        for y in x:
            print(x, end=' ')
        print(' ')

spam spam spam spam
eggs eggs eggs eggs
ham ham ham

>>> for x in ["spam", "eggs", "ham"]:
        for y in x:
            print(y, end=' ')
        print(' ')

s p a m
e g g s
h a m
>>>
for i in ["spam", "eggs", "ham"]:
        print(i, end=' ')
        print()
        for j in i:
            print(j, end=' ')
        print(' ')

spam
s p a m
eggs
e g g s
ham
h a m
>>>
```

CONSTRUCTS - B

The next two examples compute the sum and product of all the items in a list.

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
        sum = sum + x
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]:
        prod *= item
>>> prod
24
```

Any sequence works in a **for**, as it's a generic tool. For example, **for** loops work on strings and tuples:

```
>>> S = "lumberjack"
>>> for x in S:
        print(x, end=' ')
l u m b e r j a c k
>>>
>>> T = ("and", "I'm", "okay")
>>> for x in T:
        print(x, end=' ')              # Iterate over a tuple

and I'm okay
>>> words = ['apple', 'mango', 'banana', 'orange']
>>> for w in words:
        print( w, len(w))
apple 5
mango 5
banana 6
orange 6
>>> words = ['this', 'is', 'an', 'ex', 'parrot']
>>> for word in words:
        print( word)
this
is
an
ex
parrot
>>> numbers=(0,1,2,3,4,5,6,7,8,9)
>>> for number in numbers:
        print( number, end=' ')
0 1 2 3 4 5 6 7 8 9
>>>
```

CONSTRUCTS - B

USING **ELSE** WHILE REPEATING

```
> > > for food in (“pate”, “cheese”, “crackers”, “yogurt”):
        if food == “yogurt”:
            break
        else:
            print(“There is yogurt!”)

> > > for food in (“pate”, “cheese”, “crackers”):
        if food == “yogurt”:
            break
        else:
            print(“There is no yogurt!”)

There is no yogurt!
```

In each example, there is a test to determine whether there is any yogurt. If there is, the **while** ... : is terminated by using a **break** . However, in the second loop, there is no yogurt in the list, so when the loop terminates after reaching the end of the list, the **else**: condition is invoked.

There is one other commonly used feature for loops: the **continue** statement. When **continue** is used, you’re telling Python that you do not want the loop to be terminated, but that you want to skip the rest of the current statements of the loop and go back to the top, re - evaluate the conditions and the list for the next round.

USING **CONTINUE** TO KEEP REPEATING

```
> > > for food in (“pate”, “cheese”, “rotten apples”, “crackers”, “whip cream”, “tomato soup”):
        if food[0:6] == “rotten”:
            continue
        print(“Hey you can eat %s” % food)

Hey, you can eat pate
Hey, you can eat cheese
Hey, you can eat crackers
Hey, you can eat whip cream
Hey, you can eat tomato soup
```

Because you’ve used an **if ... :** test to determine whether the first part of each item in the food list contains the string “ rotten ” , the “ rotten apples ” element will be skipped by the continue , whereas everything else is printed as safe to eat.

CONSTRUCTS - B

NESTED LOOPS

Python programming language allows to use nested loops i.e., one loop inside another loop.

```
for x in range(1, 11):
    for y in range(1, 11):
        print( '%d * %d = %d' % (x, y, x*y))
```

The output of the above program would looks like the below partial output.

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
2 * 1 = 2
…
```

and so on up to

```
…
10 * 10 = 100
```

Find prime number within 30 integer numbers from 1 – 30

```
#find prime number within 30 integer numbers from 1 - 30
i = 1
while(i <= 30):
        j = 2
        while(j <= (i/j)):
                if not(i%j):
                        break
                j = j + 1
                if (j > i/j) :
                        print(i, " is prime")
        i = i + 1
print( "Good work!")
```

The output of the above program is as follows:

```
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
Good work!
```