Python for loop

A for loop is a part of a control flow statement which helps you to understand the basics of Python.

What is for loop in Python

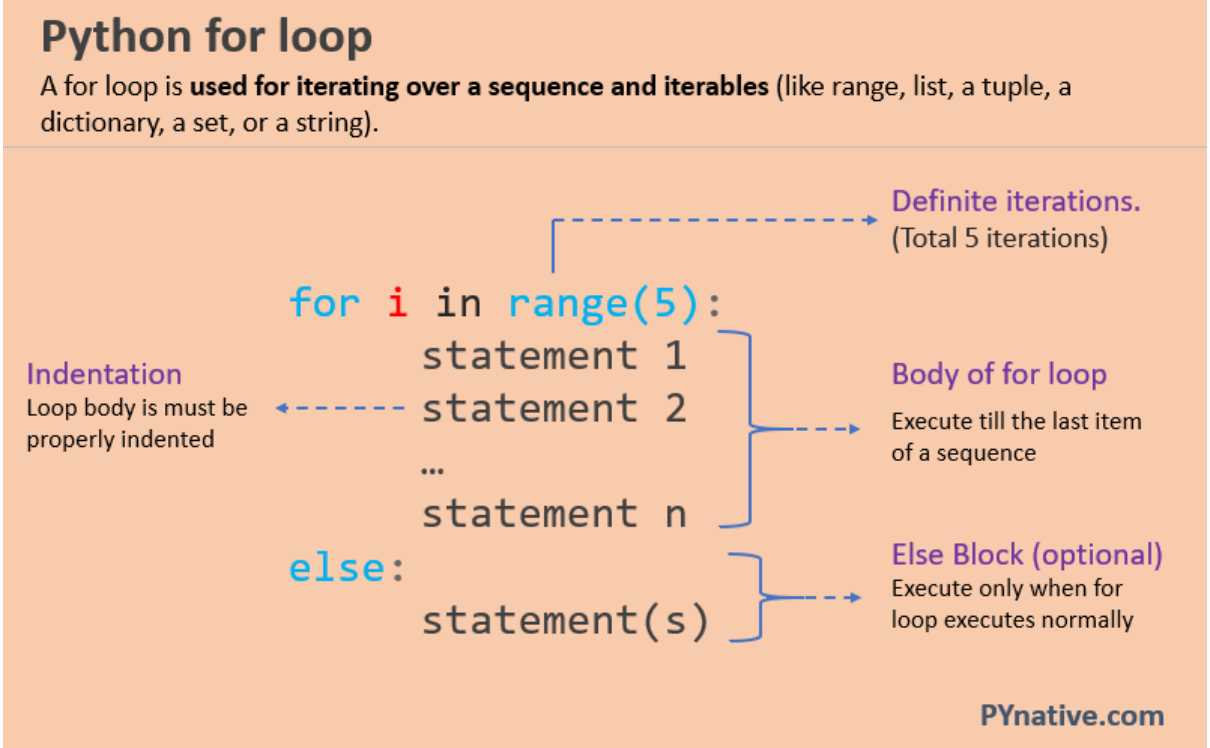
In Python, the `for` loop is used to iterate over a sequence such as a list, string, tuple, other iterable objects such as range.

With the help of `for` loop, we can iterate over each item present in the sequence and executes the same set of operations for each item. Using a `for` loops in Python we can automate and repeat tasks in an efficient manner.

So, the bottom line is using the for loop we can repeat the block of statements a fixed number of times. Let's understand this with an example.

As opposed to while loops that execute until a condition is true, `for` loops are executed a fixed number of times, you need to know how many times to repeat the code.

- **An unknown number of times**: For example, Ask the user to guess the lucky number. You don't know how many attempts the user will need to guess correctly. It can be 1, 20, or maybe indefinite. In such cases, use a `while` loop.
- **Fixed number of times**: Print the multiplication table of 2. In this case, you know how many iterations you need. Here you need 10 iterations. In such a case use `for` loop.



for loop in Python

Syntax of `for` loop

```
for i in range/sequencee:
    statement 1
    statement 2
    statement n
```

- In the syntax, `i` is the iterating variable, and the range specifies how many times the loop should run. For example, if a list contains 10 numbers, then for loop will execute 10 times to print each number.
- In each iteration of the loop, the variable `i` get the current value.

Example: Print first 10 numbers using a for loop

- Here we used the range () function to generate integers from 0 to 9
- Next, we used the `for` loop to iterate over the numbers produced by the `range ()` function
- In the body of a loop, we printed the current number.

```
for num in range (10):
    print(num)
```

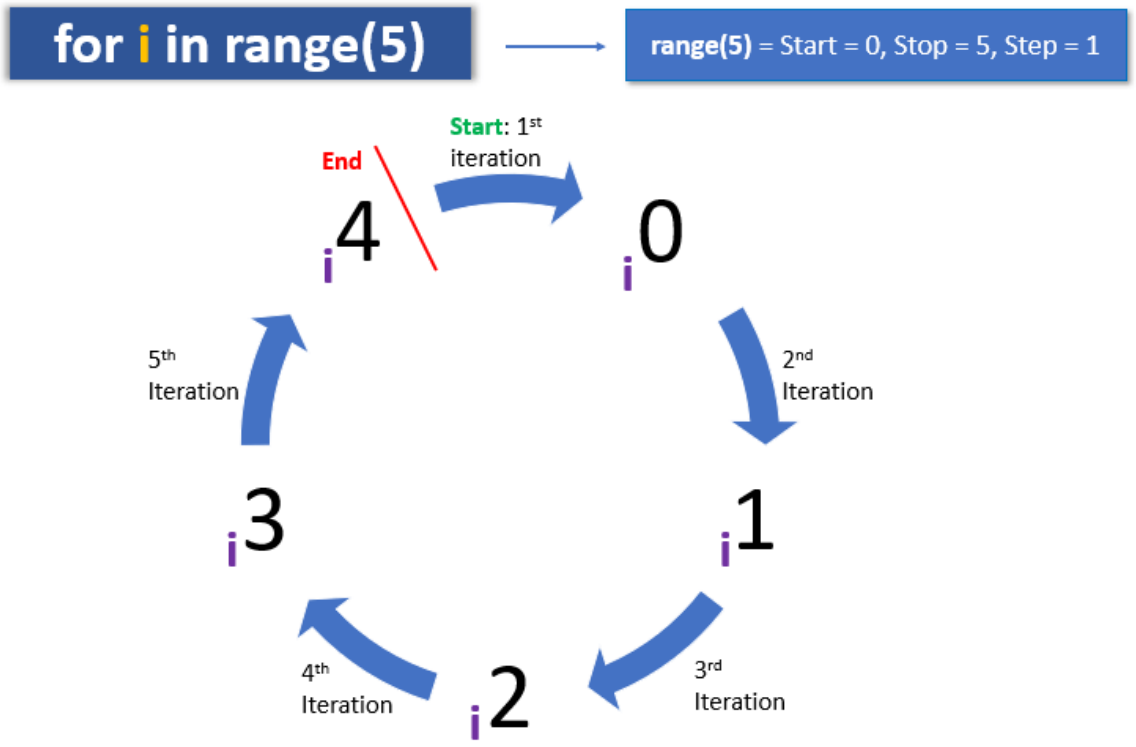
Output:

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

`for` loop with `range ()`

The range () function returns a sequence of numbers starting from 0 (by default) if the initial limit is not specified and it increments by 1 (by default) until a final limit is reached.

The `range ()` function is used with a loop to specify the range (how many times) the code block will be executed. Let us see with an example.



for loop with range ()

Example: Print sum of all even numbers from 10 to 20

- Set sum variable to zero.
- Use the `range (2, 22, 2)` to get all even numbers from 2 to 20. (Here a `step` value is 2 to get the even number because even numbers are divisible by 2)
- Next, use `for` loop to iterate over each number
- In each iteration add the current number to the sum variable using the arithmetic operator.

```
sum = 0
for i in range(2, 22, 2):
    sum = sum + i
print(sum)
# output 110
```

How `for` loop works

The `for` loop is the easiest way to perform the same actions repeatedly. For example, you want to calculate the square of each number present in the list.

Write `for` loop to iterate a list, in each iteration, it will get the next number from a list, and inside the body of a loop, you can write the code to calculate the square of the current number.

Example: Calculate the square of each number of lists

Python list is an ordered sequence of items. Assume you have a list of 10 numbers. Let's see how to want to calculate the square of each number using `for` loop.

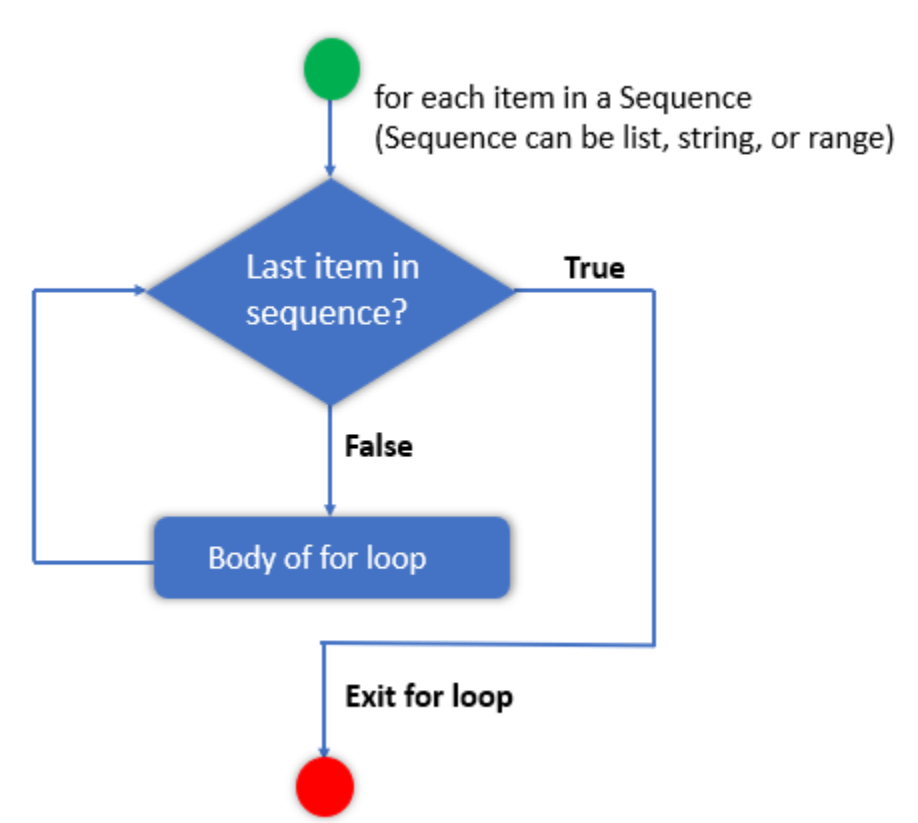
```
numbers = [1, 2, 3, 4, 5]
# iterate over each element in list num
for i in numbers:
    # ** exponent operator
    square = i ** 2
    print("Square of:", i, "is:", square)
```

Output:

```
Square of: 1 is: 1

Square of: 2 is: 4

Square of: 3 is: 9

Square of: 4 is: 16

Square of: 5 is: 25
```

Note:

The loop runs till it reaches the last element in the sequence. If it reaches the last element in the sequence, it exits the loop. otherwise, it keeps on executing the statements present under the loop's body



Flow chart of a for loop

Why use `for` loop?

Let's see the use `for` loop in Python.

- **Definite Iteration**: When we know how many times, we wanted to run a loop, then we use count-controlled loops such as for loops. It is also known as definite iteration. For example, Calculate the percentage of 50 students. here we know we need to iterate a loop 50 times (1 iteration for each student).
- **Reduces the code's complexity**: Loop repeats a specific block of code a fixed number of times. It reduces the repetition of lines of code, thus reducing the complexity of the code. Using `for` loops and `while` loops we can automate and repeat tasks in an efficient manner.
- **Loop through sequences**: used for iterating over lists, strings, tuples, dictionaries, etc., and perform various operations on it, based on the conditions specified by the user.

Example: Calculate the average of list of numbers

```
numbers = [10, 20, 30, 40, 50]

# definite iteration
# run loop 5 times because list contains 5 items
sum = 0
for i in numbers:
    sum = sum + i
list_size = len(numbers)
average = sum / list_size
print(average)
```

Output:

```
30.0
```

`If-else` in `for` loop

In this section, we will see how to use if-else statements with a loop. If-else is used when conditional iteration is needed. For example, print student names who got more than 80 percent.

The if-else statement checks the condition and if the condition is `True` it executes the block of code present inside the if block and if the condition is False, it will execute the block of code present inside the else block.

When the if-else condition is used inside the loop, the interpreter checks the if condition in each iteration, and the correct block gets executed depending on the result.

```
if condition:
    block of statements
else:
    block of statements
```

Example: **Print all even and odd numbers**

- In this program, `for` loop statement first iterates all the elements from 0 to 20.

- Next, the `if` statement checks whether the current number is even or not. If yes, it prints it. Else, the else block gets executed.

```
for i in range (1, 11):
    if i % 2 == 0:
        print ('Even Number:', i)
    else:
        print ('Odd Number:', i)
```

Output:

```
Odd Number: 1

Even Number: 2

Odd Number: 3

Even Number: 4

Odd Number: 5

Even Number: 6

Odd Number: 7

Even Number: 8

Odd Number: 9

Even Number: 10
```

Loop Control Statements in `for` loop

Loop control statements change the normal flow of execution. It is used when you want to exit a loop or skip a part of the loop based on the given condition. It also knows as transfer statements.

Now, let us learn about the three types of loop control statements i.e., `break`, `continue` and `pass`.

Break for loop

The break statement is used to **terminate the loop**. You can use the break statement whenever you want to stop the loop. Just you need to type the break inside the loop after the statement, after which you want to break the loop.

When the `break` statement is encountered, Python stops the current loop, and the control flow is transferred to the following line of code immediately following the loop.

Example: break the loop if number a number is greater than 15

- In this program, for loop iterates over each number from a list.
- Next, the if statement checks if the current is greater than 15. If yes, then break the loop else print the current number

```
numbers = [1, 4, 7, 8, 15, 20, 35, 45, 55]
for i in numbers:
    if i > 15:
        # break the loop
        break
    else:
        print(i)
```

Output:

```
1

4

7

8

15
```

Note: If the break statement is used inside a nested loop (loop inside another loop), it will terminate the innermost loop.

Continue Statement in for loop

The continue statement skips the current iteration of a loop and immediately jumps to the next iteration

Use the `continue` statement when you want to jump to the next iteration of the loop immediately. In simple terms, when the interpreter found the `continue` statement inside the loop, it skips the remaining code and moves to the next iteration.

The `continue` statement skips a block of code in the loop for the current iteration only. It doesn't terminate the loop but continues in the next iteration ignoring the specified block of code. Let us see the usage of the continue statement with an example.

Example: Count the total number of 'm' in a given string.

- In this program, the `for` loop statement iterates over each letter of a given string.
- Next, the if statement checks the current character is m or not. If it is not m, it continues to the next iteration to check the following letter. else it increments the count

```
name = "mariya mennen"
count = 0
for char in name:
    if char != 'm':
        continue
    else:
        count = count + 1

print('Total number of m is:', count)
```

Output:

```
Total number of m is: 2
```

Note: In the case of an inner loop, it continues the inner loop only.

Pass Statement in for loop

The `pass` statement is a null statement, i.e., nothing happens when the statement is executed. Primarily it is used in empty functions or classes. When the interpreter finds a `pass` statement in the program, it returns no operation.

Sometimes there is a situation in programming where we need to define a syntactically empty block. We can define that block with the pass keyword.

Let us see the usage of the pass statement with an example.

```
num = [1, 4, 5, 3, 7, 8]
for i in num:
    # Calculate multiplication in future if required
    pass
```

Else block in `for` loop

Same as the `if` statement, Python allows us to use an else statement along with for loop. In Python, for-loop can have the `else` block, which **will be executed when the loop terminates normally**. Defining the `else` part with `for` loop is optional.

else block will be skipped when

- for loop terminate abruptly
- the break statement is used to break the loop

Example 1: Else block in for loop

In this example, we are printing the first 5 numbers, and after successful execution of a loop, the interpreter will execute the else block.

```
for i in range (1, 6):
    print(i)
else:
    print("Done")
```

Output:

```
1
2
3
4
5
Done
```

Example 2: Both `break` and `else` statement

In this example, we are printing print only the first two numbers out of 5, and after that, we use the `break` statement to stop the loop. Because the loop is terminated abruptly, the else block will not execute.

```
 count = 0
for i in range (1, 6):
    count = count + 1
    if count > 2:
        break
    else:
        print(i)
else:
    print("Done")
```

Output:

```
1
2
```

Reverse for loop

Till now, we have learned about forward looping in `for` loop with various examples. Now we will learn about the backward iteration of a loop.

Sometimes we require to do reverse looping, which is quite useful. For example, to reverse a list.

There are three ways to iterating the `for` loop backward

- Reverse for loop using `range ()`
- Reverse for loop using the `reversed ()` function

Backward Iteration using the reversed () function

We can use the built-in function `reversed ()` with `for` loop to change the order of elements, and this is the simplest way to perform a reverse looping.

Example:

```
# Reversed numbers using reversed () function
list1 = [10, 20, 30, 40]
for num in reversed(list1):
    print(num)
```

Output:

```
40

30

20

10
```

Reverse for loop using range ()

We can use the built-in function `range ()` with the `for` loop to reverse the elements' order. The `range ()` generates the integer numbers between the given start integer to the stop integer.

```
Print ("Reverse numbers using for loop")
num = 5
# start = 5
# stop = -1
# step = -1
for num in (range (num, -1, -1)):
    print(num)
```

Output:

```
4
3
2
1
0
```

Example 3: Reverse a list using a loop

```
numbers = [1, 2, 3, 4]
for i in numbers[::-1]:
    print(i)
```

Output:

```
4
3
2
1
```

Nested `for` loops

Nested for loop is a `for` loop inside another `for` a loop.

A nested loop has one loop inside of another. It is mainly used with two-dimensional arrays. For example, printing numbers or star patterns. Here outer loop is nothing but a row, and the inner loop is columns.

In nested loops, the inner loop finishes all of its iteration for each iteration of the outer loop. i.e., For each iteration of the outer loop inner loop restart and completes all its iterations, then the next iteration of the outer loop begins.

Syntax of nested for loops:

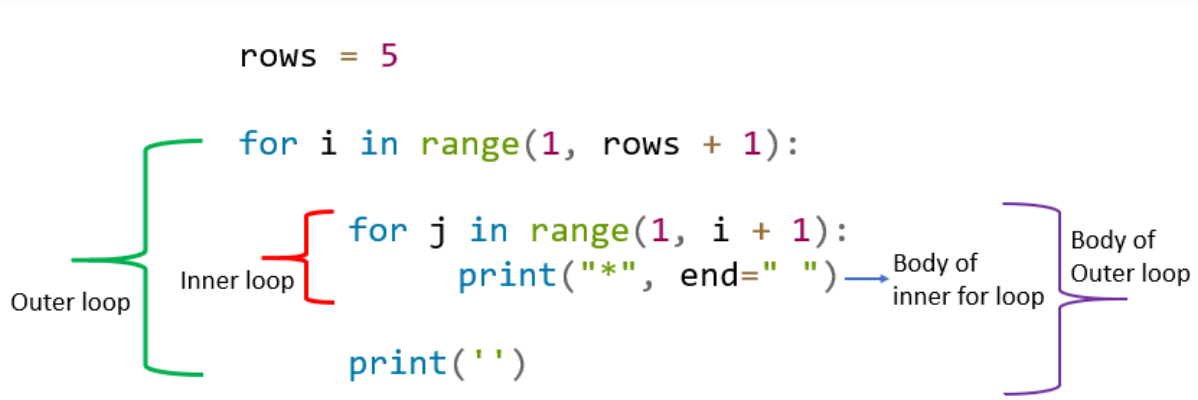
```
# outer for loop
for element in sequence
   # inner for loop
   for element in sequence:
```

```
        body of inner for loop
    body of outer for loop
other statements
```

Example: Nested for loop to print the following pattern

```
*

* *

* * *

* * * *

* * * * *
```

```
rows = 5
# outer loop
for i in range(1, rows + 1):
    # inner loop
    for j in range(1, i + 1):
        print("*", end=" ")
    print('')
```



Nested for loop

- In this program, the outer loop is the number of rows print.
- The number of rows is five, so the outer loop will execute five times
- Next, the inner loop is the total number of columns in each row.
- For each iteration of the outer loop, the columns count gets incremented by 1

- In the first iteration of the outer loop, the column count is 1, in the next it 2. and so on.
- The inner loop iteration is equal to the count of columns.
- In each iteration of an inner loop, we print star

`While` loop inside `for` loop

The while loop is an entry-controlled loop, and a `for` loop is a count-controlled loop. We can also use a `while` loop under the `for`-loop statement. Let us see an example to understand better.

Example: Print Multiplication table of a first 5 numbers using for loop and while loop

- In this program, we iterate the first five numbers one by one using the outer loop and `range` function
- Next, in each iteration of the outer loop, we will use the inner while loop to print the multiplication table of the current number

```
# outer loop
for i in range(1, 6):
    print('Multiplication table of:', i)
    count = 1
    # inner loop to print multiplication table of current number
    while count < 11:
        print(i * count, end=' ')
        count = count + 1
    print('\n')
```

`for` loop in one line

We can also formulate the `for`-loop statement in one line to reduce the number of lines of code. Let us see an example of it.

Example: Print the even numbers by adding 1 to the odd numbers in the list

```
odd = [1, 5, 7, 9]
even = [i + 1 for i in odd if i % 2 == 1]
```

```
print(even)
```

Output:

```
[2, 6, 8, 10]
```

Accessing the `index` in `for` loop

The `enumerate ()` function is useful when we wanted to access both value and its index number or any sequence such as list or string. For example, a list is an ordered data structure that stores each item with its index number. Using the item's index number, we can access or modify its value.

Using `enumerate` function with a loop, we can access the list's items with their index number. The `enumerate ()` adds a counter to iteration and returns it in the form of an enumerable object.

There three ways to access the index in for loop let's see each one by one

Example 1: Print elements of the list with its index number using the `enumerate ()` function

In this program, the for loop iterates through the list and displays the elements along with its index number.

```
numbers = [4, 2, 5, 7, 8]
for i, v in enumerate(numbers):
    print('Numbers[', i, '] =', v)
```

Output:

```
Numbers[ 0 ] = 4

Numbers[ 1 ] = 2

Numbers[ 2 ] = 5
```

```
Numbers[ 3 ] = 7

Numbers[ 4 ] = 8
```

Example 2: Printing the elements of the list with its index number using the `range()` function

```
numbers = [1, 2, 4, 6, 8]
size = len(numbers)
for i in range(size):
    print('Index:', i, " ", 'Value:', numbers[i])
```

Output:

```
Index: 0   Value: 1

Index: 1   Value: 2

Index: 2   Value: 4

Index: 3   Value: 6

Index: 4   Value: 8
```

Iterate String using `for` loop

By looping through the string using `for` loop, we can do lots of string operations. Let's see how to perform various string operations using a `for` loop.

Example 1: Access all characters of a string

```
name = "Jessa"
for i in name:
    print(i, end=' ')
```

Output:

```
J e s s a
```

Example 2: Iterate string in reverse order

```
name = "Jessa"
for i in name[::-1]:
    print(i, end=' ')
```

Output:

```
a s s e J
```

Example 3: Iterate over a particular set of characters in string

```
name = "Jessa watson"
for char in name[2:7:1]:
    print(char, end=' ')
```

Output:

```
s s a   w
```

Example 5: Iterate over words in a sentence using the `split()` function.

```
dialogue = "Remember, Red, hope is a good thing, maybe the best of things, and no
good thing ever dies"
# split on whitespace
for word in dialogue.split():
    print(word)
```

Output:

```
Remember,

Red,
```

```
hope

is

a

good

thing,

maybe

the

best

of

things,

and

no

good

thing

ever

dies
```

Iterate List using `for` loop

First, let us learn what a list is. Python list is an ordered collection of items of different data types. It means Lists are ordered by index numbers starting from 0 to the total items-1. List items are enclosed in square `[]` brackets.

Below are the few examples of Python list.

```
numers = [1,2,4,6,7]

players = ["Messi", "Ronaldo", "Neymar"]
```

Using a loop, we can perform various operations on the list. There are ways to iterate through elements in it. Here are some examples to help you understand better.

Example 1: Iterate over a list

```
numbers = [2, 3, 5, 6, 7]
for num in numbers:
    print(num)
```

Output:

```
2
3
5
6
7
```

Example 2: Iterate over a list using a for loop and range**.**

```
numbers = [1, 2, 3, 6, 7]
size = len(numbers)
for i in range(size):
    print(numbers[i])
```

Output:

```
1
2
3
6
7
```

Example 3: list comprehension

```
numbers = [1, 2, 3, 7, 8]
# list comprehension
[print(i) for i in numbers]
```

Output:

```
1
2
3
7
8
```

Iterate Dictionary using `for` loop

First, let us learn what a dictionary is. Python dictionary is used to store the items in the format of key-value pair. It doesn't allow duplicate items. It is enclosed with {}. Here are some of the examples of dictionaries.

```
dict1 = {1: "Apple", 2: "Ball", 3: "Cat"}

dict2 = {"Brand": "BMW", "Color": "Black", "Date": 1964}
```

There are ways to iterate through key-value pairs. Here are some examples to help you understand better.

Example 1: Access only the keys of the dictionary.

```
dict1 = {"Brand": "BMW", "Color": "Black", "Date": 1964}
for key in dict1:
    print(key)
```

Output:

```
Brand
Color
Date
```

Example 2: Iterate keys and values of the dictionary

```
dict1 = {"Brand": "BMW", "Color": "Black", "Date": 1964}
for key in dict1:
```

```
    print(key, "->", dict1[key])
```

Output:

```
Brand->BMW
Color->Black
Date->1964
```

Example 5: Iterate only the values the dictionary

```
dict1 = {"Brand": "BMW", "Color": "Black", "Date": 1964}
for value in dict1.values():
    print(value)
```

Output:

```
BMW

Black

1964
```