Python range() function

Python range() function returns the sequence of the given number between the given range.

range() is a built-in function of Python. It is used when a user needs to perform an action a specific number of times. range() in Python(3.x) is just a renamed version of a function called xrange in Python(2.x). The **range()** function is used to generate a sequence of numbers.

Python range() function for loop is commonly used hence, knowledge of same is the key aspect when dealing with any kind of Python code. The most common use of range() function in Python is to iterate sequence type (**Python range() List**, string, etc.) with for and while loop.

Python range syntax

range(stop)

range(start, stop[, step])

Python range() Basics

In simple terms, range() allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, user can decide where that series of numbers will begin and end as well as how big the difference will be between one number and the next.range() takes mainly three arguments.

- **start**: integer starting from which the sequence of integers is to be returned
- **stop:** integer before which the sequence of integers is to be returned. The range of integers end at stop – 1.
- **step:** integer value which determines the increment between each integer in the sequence

Example of Python range() methods

Example 1: Demonstration of Python range()

```
# Python Program to

# show range() basics


# printing a number

for i in range(10):

    print(i, end=" ")

print()


# using range for iteration
```

```
l = [10, 20, 30, 40]

for i in range(len(l)):

    print(l[i], end=" ")

print()


# performing sum of natural

# number

sum = 0

for i in range(1, 11):

    sum = sum + i

print("Sum of first 10 natural number :", sum)
```

Output :

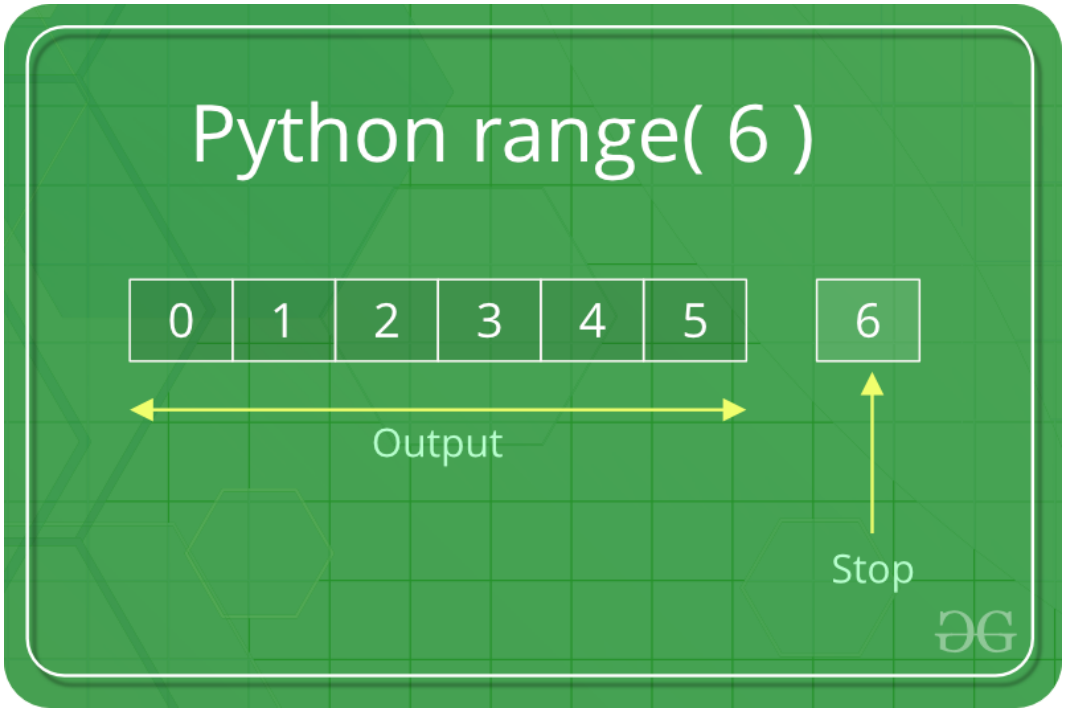
```
0 1 2 3 4 5 6 7 8 9
10 20 30 40
Sum of first 10 natural number : 55
```

There are three ways you can call range() :

- range(stop) takes one argument.
- range(start, stop) takes two arguments.
- range(start, stop, step) takes three arguments.

range(stop)

When user call range() with one argument, user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that user have provided as the stop. For Example –



Example 2: Demonstration of Python range(stop)

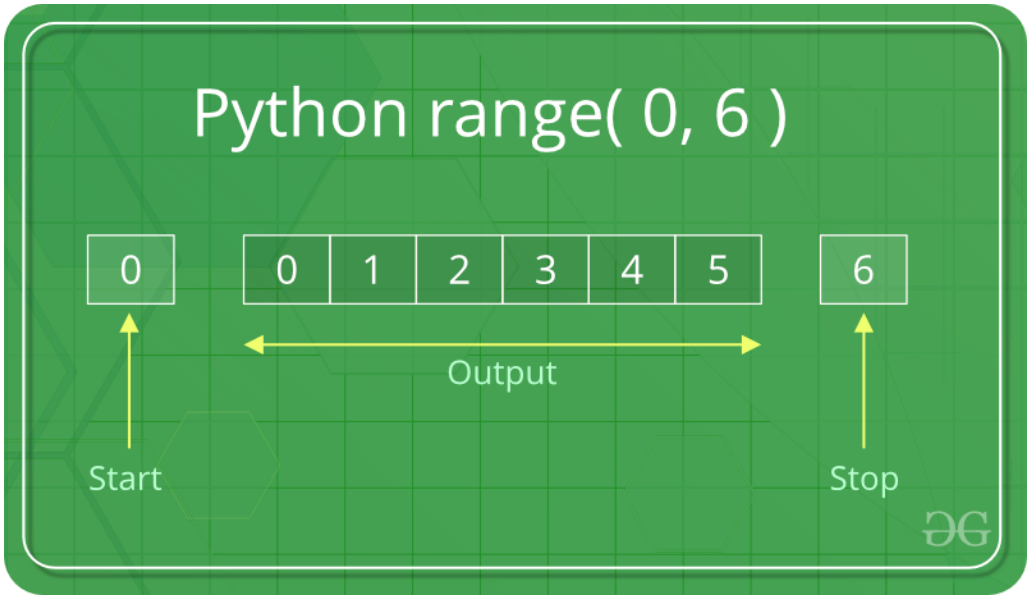
```
# Python program to

# print whole number

# using range()


# printing first 10

# whole number

for i in range(10):

    print(i, end=" ")

print()


# printing first 20

# whole number

for i in range(20):

    print(i, end=" ")
```

Output:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

range(start, stop)

When user call **range()** with two arguments, user get to decide not only where the series of numbers stops but also where it starts, so user don't have to start at 0 all the time. User can use range() to generate a series of numbers from X to Y using a range(X, Y). For Example - arguments



Example 3: Demonstration of Python range(start, stop)

```
# Python program to

# print natural number

# using range


# printing a natural

# number upto 20

for i in range(1, 20):

    print(i, end=" ")

print()


# printing a natural

# number from 5 t0 20

for i in range(5, 20):

    print(i, end=" ")
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

range(start, stop, step)

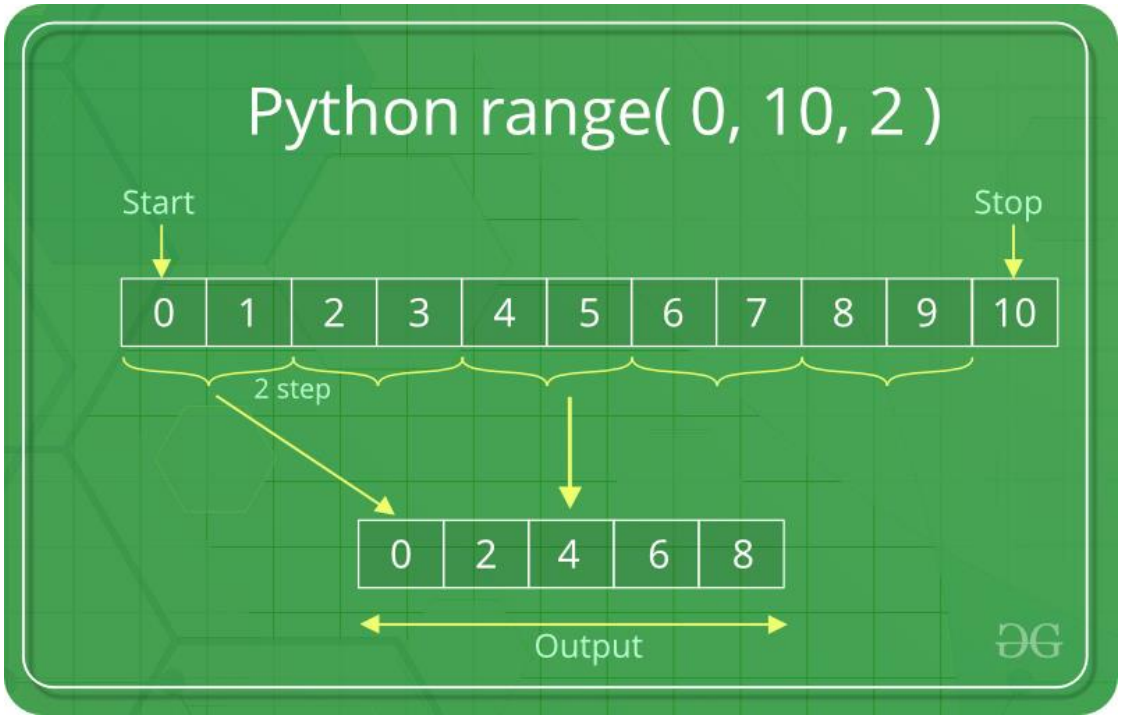
When the user call range() with three arguments, the user can choose not only where the series of numbers will start and stop but also how big the difference will be between one number and the next. If the user doesn't provide a step, then range() will automatically behave as if the step is 1.

Example 4: Demonstration of Python range(start, stop, step)

```
# Python program to

# print all number

# divisible by 3 and 5


# using range to print number

# divisible by 3

for i in range(0, 30, 3):

    print(i, end=" ")

print()


# using range to print number

# divisible by 5

for i in range(0, 50, 5):

    print(i, end=" ")
```

Output :

```
0 3 6 9 12 15 18 21 24 27
0 5 10 15 20 25 30 35 40 45
```



In this example, we are printing an even number between 0 to 10 so we choose our starting point from 0(start = 0) and stop the series at 10(stop = 10). For printing even number the difference between one number and the next must be 2 (step = 2) after providing a step we get a following output (0, 2, 4, 8).

Example 5: Incrementing with the range using positive step

If a user wants to increment, then the user needs steps to be a positive number. For example:

```
# Python program to
# increment with
# range()


# incremented by 2
for i in range(2, 25, 2):
    print(i, end=" ")
print()


# incremented by 4
for i in range(0, 30, 4):
    print(i, end=" ")
print()
```

```
# incremented by 3
for i in range(15, 25, 3):
    print(i, end=" ")
```

Output :

```
2 4 6 8 10 12 14 16 18 20 22 24
0 4 8 12 16 20 24 28
15 18 21 24
```

Example 6: Python range() backwords

If a user wants to decrement, then the user needs steps to be a negative number. For example:

```
# Python program to
# decrement with
# range()


# incremented by -2
for i in range(25, 2, -2):
    print(i, end=" ")
print()


# incremented by -4
for i in range(30, 1, -4):
    print(i, end=" ")
print()


# incremented by -3
for i in range(25, -6, -3):
    print(i, end=" ")
```

Output :

```
25 23 21 19 17 15 13 11 9 7 5 3
30 26 22 18 14 10 6 2
25 22 19 16 13 10 7 4 1 -2 -5
```

Example 7: Python range() float

Python range() function doesn't support the float numbers. i.e. user cannot use floating-point or non-integer number in any of its argument. Users can use only integer numbers. For example

```
# Python program to

# show using float

# number in range()


# using a float number

for i in range(3.3):

    print(i)


# using a float number

for i in range(5.5):

    print(i)
```

Output :

```
for i in range(3.3):
TypeError: 'float' object cannot be interpreted as an integer
```

Example 8: Concatenation of two range() functions

The result from two range() functions can be concatenated by using the chain() method of itertools module. The chain() method is used to print all the values in iterable targets one after another mentioned in its arguments.

```
# Python program to concatenate

# the result of two range functions

from itertools import chain


# Using chain method

print("Concatenating the result")

res = chain(range(5), range(10, 20, 2))


for i in res:

    print(i, end=" ")
```

Output:

```
Concatenating the result
0 1 2 3 4 10 12 14 16 18
```

Example 9: Accessing range() with index value

A sequence of numbers is returned by the range() function as its object that can be accessed by its index value. Both positive and negative indexing is supported by its object.

```
# Python program to demonstrate

# range function


ele = range(10)[0]

print("First element:", ele)


ele = range(10)[-1]

print("\nLast element:", ele)


ele = range(10)[4]

print("\nFifth element:", ele)
```

Output:

```
First element: 0

Last element: 9

Fifth element: 4
```

Points to remember about Python range() function :

- range() function only works with the integers i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a **start**, **stop** and **step** argument of a range().
- All three arguments can be positive or negative.
- The **step** value must not be zero. If a step is zero python raises a ValueError exception.
- range() is a type in Python
- Users can access items in a range() by index, just as users do with a list:

Python for loop with else

In most of the programming languages (C/C++, Java, etc), the use of else statements has been restricted with the if conditional statements. But Python also allows us to use the else condition with for loops.

Note: The else block just after for/while is executed only when the loop is NOT terminated by a break statement

```
# Python program to demonstrate
# for-else loop

for i in range(1, 4):
    print(i)
else:  # Executed because no break in for
    print("No Break\n")

for i in range(1, 4):
    print(i)
    break
else:  # Not executed as there is a break
    print("No Break")
```

Output:

```
1
2
3
No Break

1
```

While loop with else

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 your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

Note: The else block just after for/while is executed only when the loop is NOT terminated by a break statement.

```
# Python program to demonstrate

# while-else loop


i = 0

while i < 4:

    i += 1

    print(i)
```

```
else:  # Executed because no break in for
    print("No Break\n")


i = 0
while i < 4:
    i += 1
    print(i)
    break
else:  # Not executed as there is a break
    print("No Break")
```

Output:

```
1
2
3
4
No Break

1
```

Sentinel Controlled Statement

In this, we don't use any counter variable because we don't know that how many times the loop will execute. Here user decides that how many times he wants to execute the loop. For this, we use a sentinel value. A sentinel value is a value that is used to terminate a loop whenever a user enters it, generally, the sentinel value is -1.

Example: Python while loop with user input

```
a = int(input('Enter a number (-1 to quit): '))


while a != -1:
    a = int(input('Enter a number (-1 to quit): '))
```

Output:

```
Enter a number (-1 to quit): 6
Enter a number (-1 to quit): 7
Enter a number (-1 to quit): 8
Enter a number (-1 to quit): 9
Enter a number (-1 to quit): 10
Enter a number (-1 to quit): -1
```

Explanation:

- First, it asks the user to input a number. if the user enters -1 then the loop will not execute
- User enter 6 and the body of the loop executes and again ask for input
- Here user can input many times until he enters -1 to stop the loop
- User can decide how many times he wants to enter input

Looping Techniques in Python

Python supports various looping techniques by certain inbuilt functions, in various sequential containers. These methods are primarily very useful in competitive programming and also in various projects which require a specific technique with loops maintaining the overall structure of code. A lot of time and memory space is been saved as there is no need to declare the extra variables which we declare in the traditional approach of loops.

Where they are used?

Different looping techniques are primarily useful in the places where we don't need to actually manipulate the structure and order of the overall containers, rather only print the elements for a single-use instance, no in-place change occurs in the container. This can also be used in instances to save time.

Different looping techniques using Python data structures are:

- **Using enumerate():** enumerate() is used to loop through the containers printing the index number along with the value present in that particular index.

```
# python code to demonstrate working of enumerate()


for key, value in enumerate(['The', 'Big', 'Bang', 'Theory']):
    print(key, value)
```

Output:

```
0 The
1 Big
2 Bang
3 Theory
# python code to demonstrate working of enumerate()


for key, value in enumerate(['Geeks', 'for', 'Geeks',
                             'is', 'the', 'Best',
                             'Coding', 'Platform']):
    print(value, end=' ')
```

Output:

```
Geeks for Geeks is the Best Coding Platform
```

- **Using zip():** zip() is used to combine 2 similar containers(list-list or dict-dict) printing the values sequentially. The loop exists only till the smaller container ends. A detailed explanation of zip() and enumerate() can be found here.

```
# python code to demonstrate working of zip()


# initializing list
questions = ['name', 'colour', 'shape']
answers = ['apple', 'red', 'a circle']


# using zip() to combine two containers
# and print values
for question, answer in zip(questions, answers):
    print('What is your {0}?  I am {1}.'.format(question, answer))
```

Output:

```
What is your name?  I am apple.
What is your color?  I am red.
What is your shape?  I am a circle.
```

- **Using iteritem():** iteritems() is used to loop through the dictionary printing the dictionary key-value pair sequentially which is used before Python 3 version.
- **Using items():** items() performs the similar task on dictionary as iteritems() but have certain disadvantages when compared with iteritems().
    - It is **very time-consuming**. Calling it on large dictionaries consumes quite a lot of time.
    - It takes a **lot of memory**. Sometimes takes double the memory when called on a dictionary.

Example 1:

```
# python code to demonstrate working of items()


d = {"geeks": "for", "only": "geeks"}


# iteritems() is renamed to items() in python3
# using items to print the dictionary key-value pair
print("The key value pair using items is : ")
for i, j in d.items():
    print(i, j)
```

Output:

```
The key value pair using iteritems is :
```

```
geeks for
only geeks
The key value pair using items is :
geeks for
only geeks
```

Example 2:

```
# python code to demonstrate working of items()


king = {'Akbar': 'The Great', 'Chandragupta': 'The Maurya',

        'Modi': 'The Changer'}


# using items to print the dictionary key-value pair

for key, value in king.items():

    print(key, value)
```

Output:

```
Akbar The Great
Chandragupta The Maurya
Modi The Changer
```

- **Using sorted():** sorted() is used to print the **container is sorted order**. It **doesn't sort the container** but just prints the container in sorted order for 1 instance. The use of **set() can be combined to remove duplicate** occurrences.

Example 1:

```
# python code to demonstrate working of sorted()


# initializing list

lis = [1, 3, 5, 6, 2, 1, 3]


# using sorted() to print the list in sorted order

print("The list in sorted order is : ")

for i in sorted(lis):

    print(i, end=" ")


print("\r")


# using sorted() and set() to print the list in sorted order
```

```
# use of set() removes duplicates.

print("The list in sorted order (without duplicates) is : ")

for i in sorted(set(lis)):

    print(i, end=" ")
```

Output:

```
The list in sorted order is :
1 1 2 3 3 5 6
The list in sorted order (without duplicates) is :
1 2 3 5 6
```

Example 2:

```
# python code to demonstrate working of sorted()


# initializing list

basket = ['guave', 'orange', 'apple', 'pear',

          'guava', 'banana', 'grape']


# using sorted() and set() to print the list

#  in sorted order

for fruit in sorted(set(basket)):

    print(fruit)
```

Output:

```
apple
banana
grape
guava
guave
orange
pear
```

- **Using reversed():** reversed() is used to print the values of the **container in the reversed order**. It does not reflect any changes to the original list

Example 1:

```
# python code to demonstrate working of reversed()


# initializing list

lis = [1, 3, 5, 6, 2, 1, 3]
```

```
# using revered() to print the list in reversed order

print("The list in reversed order is : ")

for i in reversed(lis):

    print(i, end=" ")
```

Output:

```
The list in reversed order is :
3 1 2 6 5 3 1
```

Example 2:

```
# python code to demonstrate working of reversed()


# using reversed() to print in reverse order

for i in reversed(range(1, 10, 3)):

    print(i)
```

Output:

```
7
4
1
```

- These techniques are quick to use and reduce coding effort. for, while loops need the entire structure of the container to be changed.
- These Looping techniques do not require any structural changes to the container. They have keywords that present the exact purpose of usage. Whereas, no pre-predictions or guesses can be made in for, while loop i.e not easily understand the purpose at a glance.
- Looping technique makes the code more concise than using for & while looping.