



**Department of Computer Science and Engineering
PES University, Bangalore, India**

Lecture Notes

Python for Computational Problem Solving

UE23CS151A

Lecture #21

Control structures - Looping statements

By,

**Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU**

Verified by,

PCPS Team - 2023

Many Thanks to

**Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former
Chairperson, CSE, PES University)**

Prof. Chitra G M (Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Control Structures in Python

Looping constructs in Python

Also considered as **Iterative Control / Repetitive Control**. This is provided by a control statement that **repeatedly executes instructions** based on a given condition. In python, we have **while**, **for** and **do - while** keywords available for looping.

Let's say I want to write a program that simulates the act of waiting for the rain to stop so that I can go out. How would we wait? We would probably look outside the window periodically to see if the rain has indeed stopped.

Is it raining? Yes, then stay indoors

Is it raining? Yes, then stay indoors

Is it raining? Yes, then stay indoors

...

...

And the lines go on and on.

Finally, is it raining? No, then go out.

How can we write this in a shorter way?

Consider the looping construct **while** and the algorithm as below.

while it is still raining

stay indoors

There are **3 ingredients to any iterative statement**:

- Initial value of the iterative counter or condition
- Final Iterative condition
- Updating the iterative counter or condition in order to reach the final iterative condition

while: It is an iterative control statement that repeatedly executes a set of statements based on an expression provided. If the expression evaluates to True value, suite will be executed and control comes back to expression again. Again, suite will be executed based on the output of expression and this goes on till the expression is evaluated to False. Once the expression evaluates to False, control exits from the loop.

Syntax:

while expression: **#while is a keyword**

Statement(s) #suite of while

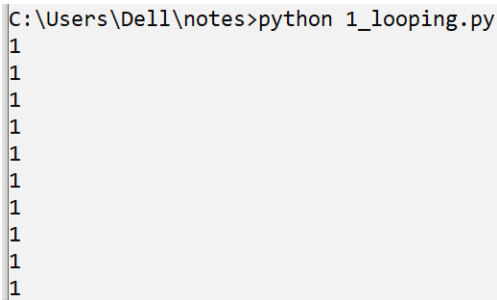
Now let us write a Program that tells us that it has stopped raining.

```
raining = True
while(raining):
    print("Stay indoors")
print("it has stopped raining, you can go out now")
```

Program 1: Write a program to print the numbers from 1 to 10 in different lines

```
i = 1
while i < 11:
    print(i)
print("end of loop")
```

Output:

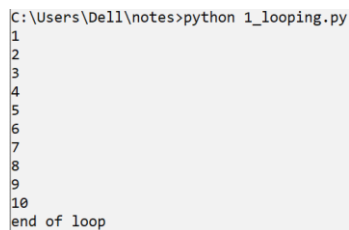


```
C:\Users\Dell\notes>python 1_looping.py
1
1
1
1
1
1
1
1
1
1
1
```

Above code results in **infinite loop (loop that never terminates or eventually terminates with a system error)** as the expression evaluates to True always.

Program 2: Modification of Program 1 to get required output.

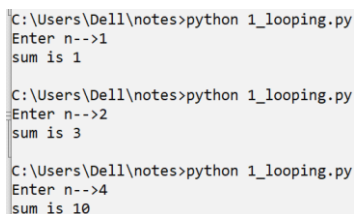
```
i = 1
while i < 11:
    print(i) # prints and increment the value of variable i by 1
    i += 1 #Shorthand operator usage
print("end of loop")
```

Output:

```
C:\Users\Dell\notes>python 1_looping.py
1
2
3
4
5
6
7
8
9
10
end of loop
```

Program 3: Program to find the sum of first n natural numbers.

```
n = int(input("Enter n-->"))
i = 1
sum = 0
while i <= n:
    sum += i
    i += 1 #i++ not available in python. No decrement and no increment operators
print("sum is", sum)
```

Output:

```
C:\Users\Dell\notes>python 1_looping.py
Enter n-->1
sum is 1

C:\Users\Dell\notes>python 1_looping.py
Enter n-->2
sum is 3

C:\Users\Dell\notes>python 1_looping.py
Enter n-->4
sum is 10
```

A few points to think about Program 3.

- Can sum be a variable?
- Can we use only sum and n avoiding the extra variable i?
- If program 3 is saved as pro.py, when we open the file in notepad++, the color of sum and the color of print is same. Any specific reason behind this?

A while statement can be used to construct both **definite and indefinite loops**.

Definite Loop

Loop in which the **number of times it iterates can be determined before the loop is executed**. Program 3 is an example code which uses the definite loop. Although it is not known what the value of n will be until the input statement is executed, its value is known by the time the while loop is reached. Thus, it will execute “n times.”

Indefinite Loop

Loop in which the **number of times it iterates cannot be determined before the loop is executed**.

Program 4:

```
inpt = input("Enter selection: ")
while inpt != 'F' and inpt != 'C':
    inpt = input("Please enter 'F' or 'C': ")
```

Output: The number of times that the loop will be executed depends on how many times the user mistypes the input.

```
C:\Users\Dell\notes>python 1_looping.py
Enter selection: e
Please enter 'F' or 'C': C

C:\Users\Dell\notes>python 1_looping.py
Enter selection: e
Please enter 'F' or 'C': b
Please enter 'F' or 'C': a
Please enter 'F' or 'C': f
Please enter 'F' or 'C': c
Please enter 'F' or 'C': F

C:\Users\Dell\notes>python 1_looping.py
Enter selection: F

C:\Users\Dell\notes>
```

for : It is an iterative control statement that **works exclusively on collections**. It examines each element and performs any action set by the programmer until there are no more elements left in the collection. Use for when the number of times we execute the body or the suite is determinable.

Syntax:

for <target_variable> in <iterable> : # for and in are keywords
<suite/body of for>

An iterable is a collection of elements physically or conceptually. It has a builtin mechanism to give an element each time we ask and it has a builtin mechanism to signal when there are no more elements.

Semantics of 'for' is as below.

1. Start iterating through the iterable.
2. Get one element to the target variable as a copy.
3. Execute the suite or the body.
4. Repeat steps 2 and 3 until the iterable signals that it has no more elements.
5. Exit the for loop and move to the next statement.

Program 5: Given the list which contains 4 elements, print the elements of the list in a separate line.

```
numbers = [34,11,23,10]
```

```
for val in numbers:
```

```
    print(val)
```

Working: The variable val gets the copy of the element 34 in the first iteration and that is displayed as part of the loop body. In the second iteration, variable val gets the element 11 and displays it. Third and forth iteration, val gets the copy of 23 and 10 respectively and displays it in the body of its corresponding iteration. When there are no more elements in the list to be copied to val, the iteration stops.

```
C:\Users\Dell\notes>python 1_looping.py
34
11
23
10
C:\Users\Dell\notes>
```

Program 6: Print the characteristics of a given string in a separate line.

```
str1 = "sindhu cse pesu"
```

```
for s in str1:
```

```
    print(s)
```

Output:

```
C:\Users\Dell\notes>python 1_looping.py
s
i
n
d
h
u

c
s
e

p
e
s
u

C:\Users\Dell\notes>
```

What if we want to make some changes to the given list in Program 5? Does the below code work? Let us try to understand in detail.

Program 7: Given a list, Increment the elements of the list by 1 and display the list.

```
numbers = [34,11,23,10]
```

```
print("Original list", numbers)
```

```
for val in numbers:
```

```
    val += 1
```

```
print("Same list after for loop execution", numbers)
```

Output: As the val variable is always a copy of one element at a time from the iterable, modifying this doesn't reflect in the given list. Refer to Program 8.

```
C:\Users\Dell\notes>python 1_looping.py
Original list [34, 11, 23, 10]
Same list after for loop execution [34, 11, 23, 10]

C:\Users\Dell\notes>
```

We will discuss the solution to this problem in Program 8 after understanding the range() function.

range: The range is a builtin function, that **generates numbers in an arithmetic progression and creates a range object.** **help(range)** is as below.

```
class range(object)
|   range(stop) -> range object
|   range(start, stop[, step]) -> range object
|
|   Return an object that produces a sequence of integers from start (inclusive)
|   to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1.
|   start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3.
|   These are exactly the valid indices for a list of 4 elements.
|   When step is given, it specifies the increment (or decrement).
```

Characteristics of range function:

- When we call / invoke the range function, it doesn't return the list itself. Rather, it **returns a conceptual collection of values.**
- **It is a Lazy function** - The values come into existence only when we explicitly ask for it. It knows what it is supposed to do, and does it only when it is forced to produce some result.
- The function **returns an iterable object** that can be used in a for loop or can be converted / extracted into a list, set or tuple

Note: Many functions producing lots of outputs are made lazy in Python 3.x. The lazy functions use less space compared to eager functions. The lazy ones may be more efficient if we do not use all the results generated by the function.

Let us use some variations of range and try understanding it.

```
>>> range(1, 10, 2)
```

```
range(1, 10, 2)
```

This output tells us that the result to call of range is an arithmetic progression with the initial value 1, the common difference 2 and the final value is less than 10. As we can see, the sequence itself is not displayed. It conceptually holds all the elements. To get all the elements from this conceptual box, iterate through using different ways as shown below.


```
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> tuple(range(1,10,2))
(1, 3, 5, 7, 9)
>>> set(range(1,10,2))
{1, 3, 5, 7, 9}
>>> for val in range(1,10,2):
...     print(val)
...
1
3
5
7
9
>>> ■
```

```
>>> list(range(5))
```

```
[0, 1, 2, 3, 4]
```

The function range **can take only one argument**. The argument is one past the final value. The default initial value is 0. The step or the common difference is 1.

```
>>> list(range(5,10))
```

```
[5, 6, 7, 8, 9]
```

The function range **can take two arguments**. The first argument is the initial value. The second argument is one past the final value. The step or the common difference is 1.

```
>>> list(range(5,15, 2))
```

```
[5, 7, 9, 11, 13]
```

The function range **can take three arguments**. The first argument is the initial value. The second argument is one past the final value. The third argument is the step or the common difference.

Few more FYA : Self explanatory.

```
>>> list(range(1,10,-1))
[]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>> list(range(1,10,0.5))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

Points to think!!

- Can we use this **range function to generate numbers** which can be used as index to the elements of the list? Yes
- Is there any **function to find the number of elements in the iterable**? Yes, the name of the function is **len()**

```
>>>numbers = [23,11,66,55]
```

```
>>>len(numbers)
```

```
4
```

Program 8: Solution to Program 7 using range() and len()

```
numbers = [34,11,23,10]
```

```
print("Original list", numbers)
```

```
for i in range(len(numbers)):
```

```
    numbers[i] += 1
```

```
print("Same list after for loop execution", numbers)
```

Output:

```
C:\Users\Dell\notes>python 1_looping.py
Original list [34, 11, 23, 10]
Same list after for loop execution [35, 12, 24, 11]

C:\Users\Dell\notes>■
```

Program 9: Program to find the square root of all numbers in separate lines from 1 to n(inclusive).

Hint: sqrt function is available in math module/library. First import math and then use sqrt with math as the module name.

Usage of functions from module/library - > object.function() or object.method()

```
import math
```

```
n = int(input("Enter the max limit: "))
```

```
for num in range(1,n+1):
```

```
    print(num, "->", math.sqrt(num))
```

Output:

```
C:\Users\Dell\notes>python 1_looping.py
Enter the max limit: 6
1 -> 1.0
2 -> 1.4142135623730951
3 -> 1.7320508075688772
4 -> 2.0
5 -> 2.23606797749979
6 -> 2.449489742783178
```

- END -