Python for loop: collection-based iteration

This type of loop iterates over a collection of objects

```
for var in iterable:
    statement(s)      #must indent
```

In Python, an *iterable* object (or simply an *iterable*) is a collection of elements that you can loop (or *iterate*) through one element at a time.
Objects like lists, tuples, dictionaries, and strings are iterable objects.

statement(s) in the loop body are denoted by indentation and are executed once for each item in *iterable*.

the loop variable *var* takes on the value of the next element in *iterable* each time through the loop.

```
L=['foo', 'bar', 'baz']

for i in L:
    print(i)
foo
bar
baz
```

Python for loop: collection-based iteration

```
for var in iterable:
    statement(s)     #must indent
```

Looping over items of a list

```
L=['foo', 'bar', 'baz']

for i in L:
    print(i)
```

Looping over characters of a string

```
s="Monty Python"

for c in s:
    print(c)
```

Looping over keys of a Dictionary

```
dzoo= {"pangolin":5, "sloth":3, "tiger":2, "turtle":10}

for k in dzoo:
    print(k,dzoo[k]) #dzoo[k] is the corresponding value
```

Dictionary methods in for loops - examples

```
dzoo= {"pangolin":5, "sloth":3, "tiger":2, "turtle":10}

dzoo.values() #generate an iterable of values
dict_values([5, 3, 2, 10])

dzoo.items() # generate an iterable of (key,value) pair tuples
dict_items([('pangolin',5), ('sloth',3), ('tiger',2), ('turtle',10)])
```

Looping over an iterable of values by using the values() method

```
for value in dzoo.values():
    print(value)
```

Looping over an itarable of key, value tuples by using the items() method

```
for key,value in dzoo.items():
    print(key,value)
```

Looping over a sequence of integer numbers: range() function

The range() function returns an object of type range, which is an iterable of a sequence of integer numbers

```
range(n)                #from 0 to n-1  and increment by 1
range(start,n)          #from start to n-1
range(start,n,step)     #from start to n-1 with increment specified by step

type(range(5))  #<class 'range'>
range(5)  #generate an iterable of sequence of numbers 0 1 2 3 4
list(range(5)) #convert to list to see the values


for i in range(5):  #0 1 2 3 4
    print(i)

for j in range(2,6): #2 3 4 5
    print(j)

for i in range(2,15,3): #2 5 8 11 14
    print(i)

for i in range(-10,-20,-2): #-10 -12 -14 -16 -18
      print(i)
```

Use of the range function in loops

- You can generate indexes within the loop to access multiple sequence types in one for loop.

```
for index in range(len(sequence):
        print(index,sequence[index])

fruits = ["apple", "banana ", "cherry"]
numbers = [30,15,25]

for i in range(len(fruits)): #loop over generated indexes
    print(i,fruits[i],numbers[i])
```

- We can use the range() function to repeat a set of code a specified number of times

```
for i in range(4):
    some=input("Enter something > ")
    print("Ah .. you entered", some)
```

Using enumerate() in for loops

enumerate(iterable, start=0) takes an iterable and adds a counter to each element, and returns an enumerated iterable object of (count, element) tuples. Count starts from 0 by default.

```
fruits = ["apple", "banana", "cherry"]

type(enumerate(fruits)) # <class 'enumerate’>
enumerate(fruits)       # <enumerate object at 0x7fdcb53747c0>
list(enumerate(fruits)) # convert to list to see the values
[(0,'apple'), (1,'banana'),(2, 'cherry')]
```

fruits

apple	banana	cherry

enumerate(fruits)

0	apple	(0,'apple')
1	banana	(1,'banana')
2	cherry	(2,'cherry')

Using enumerate() in for loops

enumerate(iterable, start=0) takes an iterable and adds a counter to each element, and retuns an enumerated iterable object of (count, element) tuples. Count starts from 0 by default.

You can use enumerate() to generate indices and access values of multiple sequence types in one for loop

```
fruits = ["apple", "banana", "cherry"]
numbers = [30,15,25]


for index, fru in enumerate(fruits):
     print(index, fru, numbers[index])
```

zip() function

zip(iterable1, iterable2, ..., iterableN) takes iterables, aggregates them, and **returns** a zip object, which is an **iterator of tuples**, where the *i*-th tuple contains the *i*-th element from each of the sequences or iterables.

```
fruits = ["apple", "banana", "cherry"]
numbers = [30,15,25]
```

`fruits`

apple	banana	cherry

`numbers`

30	15	25

`zip(numbers, fruits)`

30	apple	`(30,'apple')`
15	banana	`(15,'banana')`
25	cherry	`(25,'cherry')`

An iterator is an iterable object that can keep track of its location during iteration.

Using zip() in for loops

You can use zip() to iterate over multiple sequences in one for loop (parallel looping)

```
fruits = ["apple", "banana ", "cherry"]
numbers = [30,15,25]

zip(numbers,fruits) #<zip object at 0x7fc6802ef580>
list(zip(numbers,fruits))
[(30, 'apple'), (15, 'banana '), (25, 'cherry')]
```

```
for item1,…,itemN  in zip(iterable1,..,iterableN):
      statements
```

```
 for fru,num in zip(fruits,numbers):
       print(fru,num)
```

Using zip() in for loops

zip() provide a **safe way** to handle **iterables of unequal length**, because the iterator stops when the shortest iterable is exhausted, and the elements in longer iterables are left out.

```
fruits = ["apple", "banana ", "cherry"]
numbers = [30, 15, 25, 64, 56, 83]
colors = ["red", "yellow", "pink"]
```

You can use zip() in loops to iterate over multiple sequences of different lengths.

```
for fru, num, col in zip(fruits, numbers, colors):
      print(fru, num, col)
```

Python for loop: summary

```
for var in iterable:
     statement(s)     #must indent
```

`for item in list:`	Loop over items in a list
`for character in string:`	Loop over characters in a string
`for key in dictionary:`	Looping over the keys
`for value in dictionary.values():`	Looping over the values
`for key,value in dictionary.items():`	Looping over both keys and values
`for num in range(n):`	Loop over integer numbers
`for index in range(len(sequence)):`	Loop over indices of a sequence. Indices can be used to loop over multiple sequences.
`for index, item in enumerate(sequence):`	Loop over both indices and items of a sequence. Indices can be used to loop over multiple sequences.
`for item1,item2 in zip(iterable1,iterable2):`	Looping over items of multiple sequences at the same time

video 12-16 if-statements

Python If-statements

```
#simple if
if test:
    statements
```

If the test is True, the statements get executed
If test is False, nothing happens

```
#if-else
if test1:
    statements1
else:
    statements2
```

If the test1 is True, the statements1 get executed
If test1 is False, the statements2 get executed

```
#if-elif-else
if test1:
   statement1(s)
elif test2:
   statement2(s)
else:
   statement3(s)
```

Once a test is True, the remaining tests are not performed, and it moves to the end

Test condition syntax

Comparison Operators

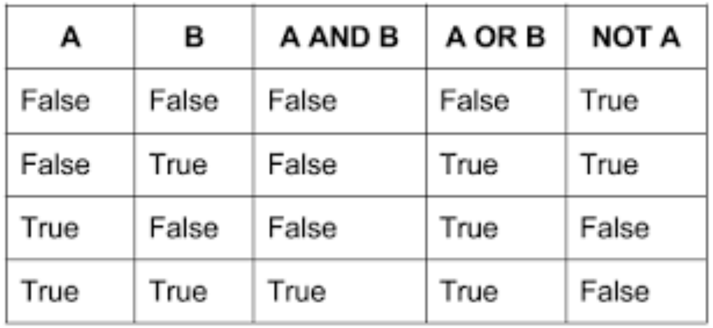
```
==   equal to
!=   not equal to

>    greater than
<    less than
>=   greater than or equal to
<=   less than or equal to
```

Membership Operators

```
member in container
member not in container
```

Logical operators

```
and
or
not
```



A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Test for membership in data types

```
member in container
member not in container
```

Test for membership in strings, lists, tuples, and dictionaries

```
#test if character(s) are in or not in a string
'p' in 'python'
'py' not in 'python'

#test if an item is in or not in a list or tuple
1 in [1,2,3]
1 not in (1,2,3)



D1={1:'a',2:'b',3:'c'}

#test if a key is in or not in a dictionary
1 in D1
1 not in D1

#test if a value is in or not in a dictionary
'a' in D1.values()
'a' not in D1.values()
```

Python If-statements examples

```
#simple if
age = 20
if age > 18:
   print("I can vote") # remember indentation


D1={1:'a',2:'b',3:'c'}
if 'p' in 'python' and 'a' in D1.values():
    print("Yes they are")


D={1:'a',2:'b'}
D1={1:'a',2:'b'}
if D==D1:
    print('They are equal’)



L=[1,2,3]
L1=[1,3,2]
if L!=L1:
    print('They are not equal')
```

Python If-statements examples

```
#if-else
color='red'
guess=input("Guess my color: ")

if color==guess:
     print("You got it")
else:
   print("Sorry")




#if-elif-else
age = 20
if age > 18:
    print("I can vote")
elif age == 18:
    print("I just turned 18 and can vote too")
else:
    print("I cannot vote")
```

The random module

```
random.random()   #return one random real number in the range [0.0,1.0)
random.randint(a,b) #return one random integer in the range[a,b]
random.choice(seq)  #return one random element from the sequence seq
random.shuffle(L)   #randomly shuffle elements of a list
```

```
import random

random.random()

random.randint(1,6)

L=['green', 'yellow', 'blue', 'orange', 'red']
s='GGCCTTCTCGAATGAATC'
```

The choice() function provides a quick way to randomly select an element from a list or a string:

```
random.choice(s)
random.choice(L)

random.shuffle(L)  #shuffle returns None
print(L)
```

https://docs.python.org/3.8/library/random.html#module-random

The while Loop

```
while test:
      statements
```

Test condition must start off as being True, and then must become false for the while loop to end.

Example:

```
num = 0
while num < 5:
    num = num + 1
    print(num)
```

Can obtain the same with a for loop

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

while loop examples

```
import random
rand_num=0

while rand_num!=8:
      print(rand_num)
      rand_num = random.randint(1,11)
```

We can also use a while True, and if-break

```
while True:
     rand_num = random.randint(1,11)
     if rand_num == 8:
        break
      print(rand_num)
```

while loop examples

```
mynumber=10
number=0   #we set this variable for the while loop to start True.

while number!=mynumber:
      number=int(input("Enter an integer number between 1-10: "))

print("You got the number: " ,number)
```

We can also use a while True, and if-break

```
mynumber=10
while True:
      number=int(input("Enter an integer number between 1-10: "))
      if number==mynumber:
         break

print("You got the number: " ,number)
```

if-break and if-continue statement in loops

The continue statement allows skipping of code within a single loop if criteria have been met. In this case if a match is found go to next entry:

```
for name in ['Newton', 'Galileo', 'Euler']:
    if 'G' in name:
        continue
    print('Hello', name)
Hello Newton
Hello Euler
```

The break statement will stop the current loop and continue with statements following the loop:

```
for name in ['Newton', 'Galileo', 'Euler']:
    if 'G' in name:
        break
    print('Hello', name)
Hello Newton
```

Summing loop: sum numbers or construct strings

```
s1 = 0                      # initiate a variable to 0
for i in range(5):          # iterate over iterable
    s1 = s1 + i              # add value and update s1
print(s1)
10
```

The same loop structure can be applied to construct a string, but with the + being the concatenation operator

```
s1 = ''                     # initiate an empty string
for i in range(5):          # iterate over iterable
    s1 = s1 + str(i)         # concatenate value and update s1
print(s1)
# str function is needed in this case, because the values in
range are not strings
01234
```

Nested loops

A nested loop is a loop inside a loop.
The "inner loop" will be executed one time for each iteration of the "outer loop":

```
for element in container:
    for element in container:
        statements(s)
    statements(s)
```

Example:

```
number=[1,2,3]
color=['blue','yellow','red']
```

For every number print each color:

```
for x in number:
    for y in color:
        print(x,y)
```