

எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு ... புரட்சிக்கவி

NOTICE

www.DataScienceInTamil.com

Day 25 - Batch 3 - Python Language

Chapter 014 List Comprehension

**To watch the recorded Python and Data Science videos in
YouTube:**

Official Website:

<https://DataScienceInTamil.com/>

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

<https://www.DatascienceInTamil.com/#faq>

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

<https://t.me/joinchat/lUZEsR-zidpjZjEx>

To Join the class, please fill the form :

<https://forms.gle/QFpLHwAoinFaX2cE6>

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

<https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVpTzZ2S1Vsdzo9>

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.
- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.
- All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content / code example, definitions, etc., please use these websites for further reading:

- Book : Python Notes For Professionals
- <https://www.w3schools.com>
- <https://www.geeksforgeeks.org>
- <https://docs.python.org>
- <https://www.askpython.com>
- <https://docs.python.org>
- <https://www.programiz.com>
- <https://www.programiz.com/>
- <https://www.openriskmanagement.com/>
- <https://pynative.com/python-sets/>
- <https://www.alphacodingskills.com/>
- <https://codedestine.com/>
- <https://appdividend.com/>
- <https://freecontent.manning.com/>

What to cover

- 1. What is Comprehensions in Python**
- 2. List comprehension**

3. Syntax for general list comprehension (Type 1)
4. Syntax for list comprehension with **if condition** (Type 2)
5. Modify only one element
6. list comprehension with **if ..else condition** (Type 3)
7. Generator Expression (tuple comprehension)
8. The expression can be as complex as needed
9. Pass argument to a function using comprehension
10. Prebuilt function inside comprehension
11. Double Iteration
12. In-place function / Mutation and Other Side Effect
13. Example for in-place function (returns None)
14. Pure function vs In-place function / Mutation
15. Whitespace in list comprehensions
16. Conditional List Comprehensions
17. Dict comprehension
18. The minimal syntax for dict comprehension
19. Conditionals in Dictionary Comprehension

- 20. Multiple if Conditional Dictionary Comprehension
- 21. if-else Conditional Dictionary Comprehension
- 22. Advantages of Using Dictionary Comprehension
- 23. Warnings on Using Dictionary Comprehension

Comprehensions in Python are **used to generate new sequences based on existing sequences**. In Python, we have comprehensions for lists, dictionaries, sets and generators. The type of the existing (original) sequence and new sequence need not be the same

Comprehensions

- 1. List comprehension []
- 2. Set comprehension {for set}
- 3. Dict comprehension {"for": "dict"}
- 4. Tuple comprehensions (Generator expressions – iter() and next())

List comprehension (reduces the lines of code)

It generate lists from other lists by applying functions to each element in the list

List comprehension is an elegant way to define and create lists

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. The following section explains and demonstrates the use of these expressions.

Common applications are to make new lists where each element is the result of some operations applied to each member of another

sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

Syntax for general list comprehension (Type 1)

```
[ <expression> for <element> in <iterable> ]
```

<expression> for <element> in <iterable>] ## the names of **expression** and the **element** must be same

Normal function

```
lst = [2,3,4,5,6,7,8,9,10,11,]  
emptyList = []  
def sample():  
    for item in lst:  
        emptyList.append(item+100)
```



```
print(emptyList)
```

```
sample()
```

output

```
[102, 103, 104, 105, 106, 107, 108, 109, 110, 111]
```

The above fn can be written used list comprehension as below

Example

```
lst = [2,3,4,5,6,7,8,9,10,11,]
```

```
sample= [(item + 100) for item in lst]
```

```
print(sample)
```

```
print(id(lst))
```

```
print(id(sample))
```

output

```
[102, 103, 104, 105, 106, 107, 108, 109, 110, 111]
```

```
36062632
```

34411272

#This is a new list, so we always have to assign to the another list

```
print(lstCom)
```

output

```
[12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
```

Syntax for list comprehension with if condition (Type 2)

```
[ <expression> for <element> in <iterable> if <condition> ]
```

use the if condition **after** for loop, if there is **no else part in the comprehension**

Example

```
lst = [2,3,4,5,6,7,8,9,10,11,]  
lstCom = [(element+100)for element in lst if element%2==0]  
print(lstCom)
```

output

```
[102, 104, 106, 108, 110]
```

Modify only one element

If we want to do operation only to one element

If we want to add 100 only to one element

```
lst = [2,3,4,5,6,7,8,9,10,11,]  
a = [ (item + 100) for item in lst if item == 4 ]  
print(a)
```

```
print(id(lst))
```

```
print(id(a))
```

output

```
[104]
```

```
31737256
```

```
30085896
```

list comprehension with if ..else condition (Type 3)

Syntax for list comprehension with if ..else condition (Type 3)

If we want to use **if and else** together in the comprehension, use it before the 'for' loop

else can be used in List comprehension constructs, but be careful regarding the syntax. **The if/else clauses should be used before for loop, not after**

Example

```
[ <expression> if <condtion>else <conditiion> for <element> in  
<iterable> ]
```

```
lst = [2,3,4,5,6,7,8,9,10,11,]
```

```
lstCom = [(element + 20)if element %2 ==0 else element%3 ==0  
for element in lst]  
print(lstCom)
```

output

[22, True, 24, False, 26, False, 28, True, 30, False]

The above code can be written as below in one line

```
print([(element + 20) if element % 2 == 0 else element % 3 == 0  
element in [2,3,4,5,6,7,8,9,10,11,]])
```

1. Note: if, if condition becomes True, the expression part / business logic part will be executed.
2. If, 'if' condition fails, the 'else' part will execute and the out will in True or False format
3. We can manipulate the else part

Another ex

```
lst = [2,3,4,5,6,7,8,9,10,11,]  
lstCom = [(element + 20) if element %2 ==0 else element + 30 for element in lst]  
print(lstCom)
```

output

[22, 33, 24, 35, 26, 37, 28, 39, 30, 41]

Note: if the remainder is 0, it adds 20, else it adds 30 to respective elements.

Another example

```
lst = [2,3,4,5,6,7,8,9,10,11,]  
lstCom = [(element + 20, element + 30) if element %2 ==0 else  
element%3 ==0 for element in lst]  
print(lstCom)
```

output

[(22, 32), True, (24, 34), False, (26, 36), False, (28, 38), True, (30, 40), False]

If we want add / modify to any one of the elements in the list, follow the below

```
lst = [2,3,4,5,6,7,8,9,10,11,]  
lstCom = [(element + 50)if element ==5 else element for element  
in lst]  
print(lstCom)
```

[2, 3, 4, 55, 6, 7, 8, 9, 10, 11]

```
lst = [21,31,41,51,61,71,81,91,101,111,]  
lstCom = [(element)if element else element [1]+100 for element  
in enumerate(lst)]  
print(lstCom)
```



```
print("-----")
for element in enumerate(lst):
    print(element)
    print(element[1]+100)
    print("-----")
```

output

```
[(0, 21), (1, 31), (2, 41), (3, 51), (4, 61), (5, 71), (6, 81), (7, 91), (8,
101), (9, 111)]
```

```
-----
```

```
(0, 21)
```

```
121
```

```
-----
```

```
(1, 31)
```

```
131
```

```
-----
```

(2, 41)

141

(3, 51)

151

(4, 61)

161

(5, 71)

171

(6, 81)

181

(7, 91)

191

(8, 101)

201

(9, 111)

211

Above code can be written using fn

```
def comprehension(lst):  
    list_comp = []  
    for x in lst:  
        if x % 2 == 0:  
            list_comp.append(x + 20)  
        else:
```

```
list_comp.append(x % 3 == 0)
return list_comp
```

```
print(comprehension([2, 3, 4, 5, 6, 7, 8, 9, 10, 11]))
```

The expression can be as complex as needed:

The expression applied to each element can be as complex as needed:

Get a list of uppercase characters from a string

```
my_list = "Hello World" # Watch, this is string
my_list = [ ( s.upper() ) for s in my_list ]
print (my_list)
```

```
my_list = ["Hello World"] # watch, this is a list with one element
```

```
my_list=[(s.upper()) for s in my_list]  
print (my_list)
```

output

```
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```

```
['HELLO WORLD']
```

Note: But this is not that we intended. It happens because we did not give the iterable inside comprehension. See the next program

The above pgm should be as below

```
my_list = ["Hello World", "Muthu"]
```

```
my_list=[(s.upper()) for s in my_list] # the expression is inside the  
tuple
```

```
print (my_list)
```

output

```
['HELLO WORLD', 'MUTHU']
```

```
my_list=[s.upper() for s in "Hello world"] # the iterable in side the  
comprehension
```

```
print (my_list)
```

output

```
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D'] # the reason, the iterable  
is given inside the comprehension. this is RIGHT way of doing
```

```
my_list = ["Hello World", "Muthu"]
```

```
my_list=[s.upper() for s in my_list[0]] # index based upper case
```

```
print (my_list)
```

output

```
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```

Slicing

```
my_list = ["Hello World", "Muthu", "Valli", "Lognathan"]  
my_list=[s.upper() for s in my_list[0:3]]  
print (my_list)
```

output

```
['HELLO WORLD', 'MUTHU', 'VALLI']
```

We can pass argument to a function using comprehension

```
def calculateSalary(hourly):  
    ratePerHr = 8  
    return ratePerHr * hourly
```

```
my_list = [10,20, 30]  
wages = [calculateSalary(i) for i in my_list] # ## the names of
```

expression (word) and the element must be same (word)

```
print(wages)
```

output

```
[80, 160, 240]
```

=====

Use prebuilt function inside comprehension

Strip off any commas from the end of strings in a list


```
my_list = ['these,', 'words,,,', 'mostly', 'have, commas,']  
my_list = [word.strip(',') for word in my_list] # ## the names of  
expression (word) and the element must be same (word)  
print(my_list)
```

output

```
['these', 'words', 'mostly', 'have, commas']
```

If we don't give any separator value inside the strip(), it prints the list as it is. See below

```
my_list = ['these,', 'words,,,', 'mostly', 'have, commas,']  
my_list = [word.strip("") for word in my_list] # strip function does  
not have any strip value to be stripped  
print(my_list)
```

output

```
['these,', 'words,,,', 'mostly', 'have, commas,']
```

=====

Organize letters in words more reasonably - in an alphabetical order

```
def split(self,  
    sep: Optional[str] = ...,  
    maxsplit: int = ...) -> List[str]
```

Return a list of the words in the string, using sep as the delimiter string.

sep

The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit

Maximum number of splits to do. -1 (the default value) means no limit.

=====

Practice the below individual programs before going to next pgm

```
sentence = "Beautiful"
```

```
b = list(sentence)
```

```
print(b)
```

output

```
['B', 'e', 'a', 'u', 't', 'i', 'f', 'u', 'l']
```

=====

```
sentence = "Dusky skin is better than white skin"
```

```
sentence=[word for word in sentence.split()]
```

```
print(sentence)
```

['Dusky', 'skin', 'is', 'better', 'than', 'white', 'skin']

```
# =====
```

```
sentence = ['Dusky', 'skin', 'is', 'better', 'than', 'white', 'skin']
```

```
sentence = " ".join(sentence)
```

```
print(sentence)
```

```
print(type(sentence))
```

```
=====
```

```
sentence = "Beautiful is better than "
```

```
sentence = [word for word in sentence.split()]
```

```
# sentence = [(sorted(word, key=lambda x: x)) for word in  
sentence.split()]
```

```
print(sentence)
```

```
# output: ['Beautiful', 'is', 'better', 'than', 'ugly']
```

```
print(" ".join(sentence)) # join(), again join all the list items to  
a string
```

```
# outout: Beautiful is better than ugly
```

```
=====
```

Check if a character is available in a string

```
a= 'apple'
```

```
lst = list(a)
```

```
b = (lst)
```

```
print(b)
```

```
for char in b:
```

```
    if char in ['a','e', 'i', 'o', 'u']:
```

```
        print("Yes")
```

```
    else:
```

```
        print("No")
```

output

```
['a', 'p', 'p', 'l', 'e']
```

```
Yes
```

```
No
```

```
No
```

No

Yes

=====

How to print non vowels

```
st = 'apple'
```

```
b = list(st)
```

```
for item in b:
```

```
    if item not in ['a', 'e', 'i', 'o', 'u']:
```

```
        print(item)
```

output

p

p

l

Find a non-vowles from a string and replace with *

```
str = "apple"  
lst1 = list(str)  
print(lst1)  
emptyList = []  
for char in lst1:  
    if char in ['a','e','i','o','u']:  
        emptyList.append(char)  
        # print(emptyList)  
    else:  
        emptyList.append('*')  
print(emptyList)
```

output

['a', 'p', 'p', 'l', 'e']

['a', '*', '*', '*', 'e']

=====

create a list of characters in apple, replacing non vowels with '*'

Ex - 'apple' --> ['a', '*', '*', '*', 'e']

=====

replaceVowel = [(X) **if** X **in** 'aeiou' **else** '*' **for** X **in** 'apple'] # the names of expression and the element must be same. Here the expression is

 # X **if** X **in** 'aeiou' **else** '*' , element is X
print (replaceVowel)

output

['a', '*', '*', '*', 'e']

IMPORTANT:Note this uses a different language construct, a conditional expression, which itself is not part of the

comprehension syntax. Whereas the **if after the for...in is a part of list comprehensions** and used to **filter elements** from the source iterable.

=====

```
lst = ['a', 'p', 'p', 'l', 'e']
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

```
listCom = [(element) if element in vowels else '*' for element in  
lst]
```

```
print(listCom)
```

output

```
['a', '*', '*', '*', 'e']
```

=====

Double Iteration

Order of double iteration [... **for x in ... for y in ...**] is either natural or counter-intuitive. The rule of thumb is to **follow an equivalent for loop**

```
def foo (i):  
    return i, i + 0.5
```

```
for i in range(3):  
    result = foo(i)  
    print(result)
```

```
print("=====")
```

```
def foo (i):  
    yield i, i + 0.5
```

```
for i in range(3):  
    result = foo(i)  
    print(next(result))
```

Above code can be written using list comprehension as below

```
def foo (i):  
    return i, i + 0.5  
  
result = [ (foo(item))          for item in range(3)          ]  
print(result)
```

output

```
[(0, 0.5), (1, 1.5), (2, 2.5)]
```

=====

The above code can be re written in order to get the result – see below

```
def foo (i):  
    return i, i + 10  
  
for i in range(3):  
    for x in foo(i):  
        print (str(x)) # or print(x,i)
```

output

0

10

1

11

2

12

=====

The above code can be re written as below

```
def fn1(value):
```

```
    # print(value+ 100)
```

```
    return value, value * 10
```

```
for item in range(1,5):
```

```
a = fn1(item)
```

```
print(a)
```

output

```
(1, 10)
```

```
(2, 20)
```

```
(3, 30)
```

```
(4, 40)
```

the above code can be written in one line as list comprehension as below

```
def foo (i):
```

```
    return i, i + 10
```

```
listCom = [str (x) for i in range(3) for x in foo(i) ]
```

```
print(listCom)
```

output

```
['0', '10', '1', '11', '2', '12']
```

Note: observe the str(x) in the comprehension, if we removed the str. It gives numerical list [0, 10, 1, 11, 2, 12]

Above code can be written in as below using list comprehension

```
def fn1(value):  
    # print(value+ 100)  
    return value, value * 10
```

```
Abi = [ (fn1(item))          for item in range(1,5) ]  
print(Abi)
```

output

```
[(1, 10), (2, 20), (3, 30), (4, 40)]
```

Nested for loop and List Com

Idea has been given by Abinaya (nested for loop in list comprehension)

```
def fn1(value):  
    return value, value + 10
```

```
def fn2(value):  
    return value, value + 100
```

```
Abi = [ (fn1(item), fn2((item1)))  
for item in range(1,5) for item1 in range(1,6) ]  
for value in Abi:  
    print(value)
```

result

run the code to see the result

=====

Pure function vs In-Place function

In-place function / Mutation and Other Side Effect

`sort ()` is an example for an **in-place** function, because it modifies content of the list, but gives `None`, where a **pure** function modifies the content but gives output

Before using list comprehension, understand the difference between functions called for their side effects (mutating, or **in-place** functions) which usually **return None**, and **functions that return an interesting value**.

Many functions (especially **pure** functions) simply take an object and **return some object**.

An in-place function modifies the existing object, which is called a **side effect**. Other examples include input and output operations such as printing.

Example for in-place function (returns None)

```
lst = [ [2, 1, ], [4, 3, ], [0, 1, ] ]  
a = lst.sort()  
print(a)
```

output

None

How it sorts

```
lst = [  
    [2, 4, ],  
    [2, 3, ],  
    [0, 1, ]  
]  
lst.sort()  
print(lst)  
output  
[[0, 1], [2, 3], [2, 4]]
```

First it sorts the first col 0 2 2 then second col 1 3 4

Note . Sorting multiple dim also possible using sort()

```
lst = [ [2, 2, [1,3] ], [2, 2, [1,4]], [2, 2, [0,2]] ]  
lst.sort()  
print(lst)
```

output

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

=====

The above program can be written using comprehension

```
my_list = [lstA.sort() for lstA in [ [2, 1, ], [4, 3, ], [0, 1, ] ] ]  
print (my_list)
```

output

```
[None, None, None]
```

```
lstA= [ [2, 1, [6,9]], [4, 3,[2,8]], [0, 1,[7,9]] ]
```

```
print(lstA)
```

```
lstA.sort()
```

```
print(lstA)
```

output

```
[[2, 1, [6, 9]], [4, 3, [2, 8]], [0, 1, [7, 9]]]
```

```
[[0, 1, [7, 9]], [2, 1, [6, 9]], [4, 3, [2, 8]]]
```

Note, It requires to sort the first col (0,2,4)

Instead of getting the output None, if we want this list to be sorted, use the below pgm

```
lst = [ [2, 1, ], [4, 3, ], [0, 1, ] ]  
mylist = [a.sort() for a in lst]  
print(mylist)  
print(lst)
```

output

```
[None, None, None]
```

```
[[1, 2], [3, 4], [0, 1]] # take each sub list then sort it, then go to  
next sub list
```

sort() // sort () is an example for an **in-place** function, because it modifies content of the list, but gives None, where a **pure** function modify the content but gives modified output

the above program can be written as below using inside a function (functions that return an interesting value.

```
def method1(lstA):  
    lstA.sort()  
    print ("after sort The values are :", lstA.sort()) # if we print  
here, it gives the EXPECTED result , is None  
    return lstA
```

```
lstB = [ [2, 1, ], [4, 3, ], [0, 1, ] ]  
a= method1(lstB) # calling the function method1 and and stores  
returns value. It supposed to be the value after applying the  
sort(), that is None
```

`print (a)` # but we get the list it self, which is unexpected

output

after sort The values are : None

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

Note: the above the id is going to be same. Cos' it is sorting but the memory location is not changed. Check with id

```
def m(lst):  
    return lst.sort()
```

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

```
a = m(lst)
```

```
print(a)
```

output

None

Pure function vs In-place function / Mutation

Example for pure function (returns some object)

`sorted()` returns a sorted list rather than sorting in-place:

Sorted ()

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

```
a= sorted(lst)
```

```
print (a)
```

output

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

Using comprehensions for side-effects is possible, such as I/O or in-place functions. Yet a for loop is usually more readable. While this works in Python 3

```
[print (x) for x in (1, 2, 3,)]
```

Instead

```
for x in (1,2,3):  
    print (x)
```

Both give same output, still use of the **for loop** is preferable

Amother ex

```
a = [ print (x) for x in (1, 2, 3,)]  
print(a)
```

output

1

2

3

[None, None, None]

=====

In some situations, side effect functions (in-place) are suitable for list comprehension. `random.randrange()` has the side effect of changing the state of the random number generator, but it also returns an interesting value. Additionally, `next()` can be called on an iterator

The following random value generator is not pure, yet makes sense as the random generator is reset every time the expression is evaluated:

`print(ra.randrange(0))` is not valid. It must be >0 and positive integer

`print(ra.randrange(-5))` is not valid. It must be >0 and positive integer

`print(ra.randrange(5))` is valid

see the same above pgm in list comprehension below

```
a = [ ( ra.randrange(5) )  for item in range(1,10) ]  
print(a)
```

output

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

The below will not work as the randrange STARTS from 0

```
a = [ ( ra.randrange(0) )  for item in range(1,10) ]  
print(a)
```

output is error

```
raise ValueError("empty range for randrange()")
```

```
ValueError: empty range for randrange()
```

`random.randrange (stop)`

`random.randrange (start, stop [, step])`

Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

The positional argument pattern matches that of `range()`. Keyword arguments should not be used because the function may use them in unexpected ways

The following random value generator is not pure fn, yet makes sense as the random generator is reset every time the expression is evaluated:

```
a= [random.randrange (1, 7) for _ in range(10)] # LOOK : The _  
(hypothesis), what does it do, it is iteration variable  
print (a)
```

output

[2, 3, 2, 5, 4, 6, 1, 5, 2, 1] # it prints 10 random numbers
between 1 and 7

NOTE: randrange(), includes start and excludes the stop

```
a= [random.randint (1, 7) for _ in range(10)] # : What does the _  
(hypothesis) do  
print (a)
```

output

[3, 7, 7, 3, 4, 2, 3, 7, 6, 2]

NOTE: randint(), includes start and the stop

How the step in randomrange works randomrange(start, stop, step)

First it prints the start number, the start number + step, then it keeps add the step number

Here first it prints 1

Then it adds $1 + 3$ (step numner) = 4

Then $4 + 3$ step number = 7

And so on..

```
from random import randrange
```

```
a= [randrange (1,10,3) for _ in range(10)]
```

```
print (a)
```

output

```
[1, 4, 7, 4, 4, 7, 7, 7, 1, 1]
```

Whitespace in list comprehensions

More complicated list comprehensions can reach an undesired length, or become less readable. Although less common in examples, it is possible to break a list comprehension into multiple lines like so:

```
a=[x for x in 'foo' if x not in 'bar']
```

```
print (a)
```

output

```
['f', 'o', 'o']
```

For readability / whitespace purpose, the above pgm should have been written as below

```
a=[
    x for x
    in 'foo'
    if x not in 'bar'
]
print (a)
```

output

```
['f', 'o', 'o']
```

```
a = [      ( item1 for item1 in item*10 )      for item in "Muthu"
]
b = ((a))
```

```
for item in b:  
    print(list(item))
```

```
# for item in 'Muthu':  
#     print(item * 10)
```

output

```
['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M']
```

```
['u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u']
```

```
['t', 't', 't', 't', 't', 't', 't', 't', 't', 't']
```

```
['h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h', 'h']
```

```
['u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u']
```

Another example


```
lst1 = [1,2,3,4]
```

```
lst2 = [1,6,7,8]
```

```
a = [val for val in lst1 if val in lst2]  
print(a)
```

```
a = [val for val in lst1 if val not in lst2]  
print(a)
```

output

```
[1]
```

```
[2, 3, 4]
```

```
=====
```

Conditional List Comprehensions

List slicing (selecting parts of lists)

Slicing does not have **from** and **to** index

Section 22.1: Using the third **"step"** argument

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

```
print (lst[::2])
```

"IT PRINTS THE FIRST CHAR THEN 1 + 2 + 2 AND SO ON"

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

```
print (lst[::3])
```

"IT PRINTS THE FIRST CHAR THEN 1 + 3 + 3 AND SO ON"

OUTPUT

```
['a', 'c', 'e', 'g']
```

```
['a', 'd', 'g']
```

You can use a list comprehension for comparing two lists element wise then use [all](#) function to check all of comparison are True

```
a = [8,9,9,11]
b = [8,7,20,10]
x = [a[i] >= b[i] for i in range(len(a))]
print(x)
```

output

```
[True, True, False, True]
```

```
a = [8,9,9,11]
b = [8,7,20,10]
print(all([a[i] >= b[i] for i in range(len(a))]))
print(any([a[i] >= b[i] for i in range(len(a))]))
```

output

False

True

Using not all and not any

```
a = [8,9,9,11]
b = [8,7,20,10]
print(not all([a[i] >= b[i] for i in range(len(a))]))
print(not any([a[i] >= b[i] for i in range(len(a))]))
```

True

False

Dict comprehension

Dictionaries are data types in Python which allows us to store data in **key/value pair**. For example:

```
my_dict = {1: 'apple', 2: 'ball'}
```

Normal for loop with range()

```
square_dict = dict()
for num in range(1, 11):
    square_dict[num] = num*num
print(square_dict)
```

output

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

What is Dictionary Comprehension in Python?

Dictionary comprehension is an elegant and concise way to create dictionaries.

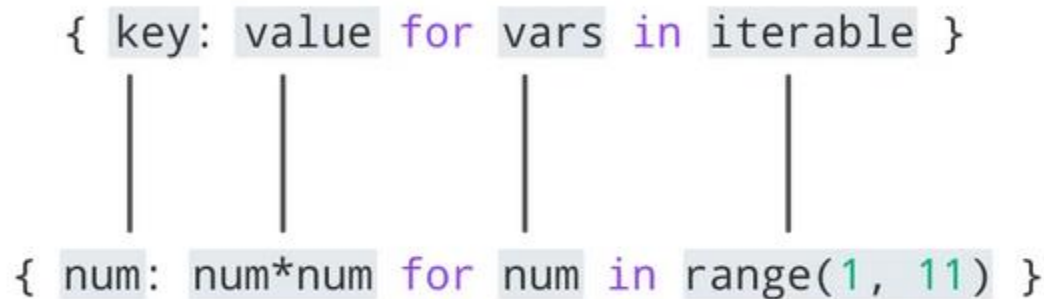
Above can be written in dict comprehension

```
square_dict = {num: num*num for num in range(1, 11)}
print(square_dict)
```

Note: using dictionary comprehension allowed us to **create a dictionary in a single line**.

The minimal syntax for dictionary comprehension is:

dictionary = {key: value for vars in iterable}



The diagram illustrates the components of a dictionary comprehension. It shows a general syntax line: { key: value for vars in iterable }. Below this, four vertical lines connect each part to a corresponding part in a specific example line: { num: num*num for num in range(1, 11) }. The connections are: 'key' to 'num', 'value' to 'num*num', 'vars' to 'num', and 'iterable' to 'range(1, 11)'.

```
{ key: value for vars in iterable }
```



```
{ num: num*num for num in range(1, 11) }
```

how we can use dictionary comprehension using data from another dictionary? – see below code

```
old_price = {'milk': 1.2, 'coffee': 2.5, 'bread': 2.5}
```

```
dollar_to_pound = 50
```

```
new_price = {item: value*dollar_to_pound for (item, value) in  
old_price.items()}  
print(new_price)
```

output

```
{'milk': 60.0, 'coffee': 125.0, 'bread': 125.0}
```

Using dictionary comprehension makes this task much simpler and shorter.

Conditionals in Dictionary Comprehension

(Assignment : do the below in normal way / using function)

We can further customize dictionary comprehension by adding conditions to it.
Let's look at an example.

```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
```

```
even_dict = {k: v for (k, v) in original_dict.items() if v % 2 == 0}  
print(even_dict)
```

output

```
{'jack': 38, 'michael': 48}
```

Note: only the items with even value have been added, because of the `if` clause in the dictionary comprehension.

Another code to extract only the values and then use %

```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 34}  
even_dict = { (data) for data in original_dict.values() if data % 2 == 0 }  
print(even_dict)
```

output

```
{48, 34, 38}
```

Multiple if Conditional Dictionary Comprehension

(Assignment : do the below in normal way / using function)

```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
```

```
new_dict = {k: v for (k, v) in original_dict.items() if v % 2 != 0 if v < 40}  
print(new_dict)
```

output

```
{'john': 33}
```

if-else Conditional Dictionary Comprehension

(Assignment : do the below in normal way / using function)

```
student_mark_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
```

```
reuslt_dict = {k: ('FAIL' if v > 40 else 'PASS')    for (k, v) in  
student_mark_dict.items()}  
print(reuslt_dict)
```

output

```
{'jack': 'PASS', 'michael': 'FAIL', 'guido': 'FAIL', 'john': 'PASS'}
```

Find inventory using dict comprehension

(Assignment : do the below in normal way / using function)

```
student_mark_dict = {'Laptop': 0, 'Mobile': 48, 'guitar': 57, 'Hardisk': 0}
```

```
reuslt_dict = {k: ('Stock' if v > 0 else 'No Stock')    for (k, v) in  
student_mark_dict.items()}  
print(reuslt_dict)
```

output

```
{'Laptop': 'No Stock', 'Mobile': 'Stock', 'guitar': 'Stock', 'Hardisk': 'No Stock'}
```

Advantages of Using Dictionary Comprehension

As we can see, dictionary comprehension shortens the process of dictionary initialization by a lot. It makes the code more pythonic.

Using dictionary comprehension in our code can shorten the lines of code while keeping the logic intact.

Warnings on Using Dictionary Comprehension

Even though dictionary comprehensions are great for writing elegant code that is easy to read, they are not always the right choice.

We must be careful while using them as :

- They can sometimes make the code run slower and consume more memory.
- They can also decrease the readability of the code.

We must not try to fit a difficult logic or a large number of dictionary comprehension inside them just for the sake of making the code single lined. In these cases, It is better to choose other alternatives like loops.

=====

Tuple comprehension

Generator Expression

```
lst = [21,30,41,51,60,71,81,91,101,111,]  
generatorExpression = ((element + 20) for element in lst)  
print(generatorExpression)  
print(next(generatorExpression))  
print(list(generatorExpression))
```

output

```
<generator object <genexpr> at 0x01BF7798>
```

```
41
```

```
[50, 61, 71, 80, 91, 101, 111, 121, 131]
```

Another example

```
lst = [21,30,41,51,60,71,81,91,101,111,]  
generatorExpression = ((element + 20)if element %2 ==0 else  
element+30 for element in lst)
```

```
print(generatorExpression)  
print(next(generatorExpression))  
print(next(generatorExpression))  
print(next(generatorExpression))  
print(next(generatorExpression))
```

output

```
<generator object <genexpr> at 0x01AA8808>
```

```
51
```

```
50
```

71

81

Each <element> in the <iterable> is plugged in to the <expression> if the (optional) <condition> evaluates to true . **All results are returned at once in the new list.**

Generator expressions are evaluated lazily, but list comprehensions evaluate the entire iterator immediately - consuming memory proportional to the iterator's length

The for expression sets x to each value in turn from (1, 2, 3, 4). **The result of the expression $x * x$ is appended to an internal list.** The internal list is assigned to the variable my_list when completed.

Besides a speed increase (as explained here), a list comprehension is roughly equivalent to the following for-loop:

```
squares = [ ]  
for x in (1, 2, 3, 4, 5):  
    squares.append(x*x)  
    print (x)  
print (squares)
```

output

1

2

3

4

5

[1, 4, 9, 16, 25]

=====

List comprehension does not accept if, elif, else. it accepts only one if and else.

to use if, elif, else, follow the idea

```
def my_generator():  
    for x in range(10):  
        if(x%2 == 0):  
            yield x  
        elif(x < 5):  
            yield x+10  
        else:  
            yield x+100
```

```
result = my_generator()
```



```
print(resultt)
print(next(resultt))
print(next(resultt))
print(next(resultt))
print(next(resultt))
print(next(resultt))
print(next(resultt))
```

output

<generator object my_generator at 0x000001E2EF349BD0>

0

11

2

13

4

105

We can pass the generator object to the list comprehension – see below

```
def my_generator():  
    for x in range(10):  
        if (x%2 == 0):  
            yield x  
        elif (x < 5):  
            yield x+10  
        else:  
            yield x+100
```

```
resultt = my_generator()  
print(resultt)
```

```
my_list = [(item) for item in my_generator()] # or use my_list = [(item) for item in  
my_generator()]  
print(my_list)
```

output

<generator object my_generator at 0x000002045CF99AF0>

[0, 11, 2, 13, 4, 105, 6, 107, 8, 109]

Generator expression

```
my_list = ["Hello World", "Muthu"]  
my_list=( (s.upper()) for s in my_list[1] )  
print (my_list)  
print(next(my_list))  
print(next(my_list))  
print(next(my_list))
```

output

<generator object <genexpr> at 0x01DE77D0>

M

U

T
