**List Methods**

List Comprehension

**--** **How to create a list in python?**

1. Using For Loop:

-- The most common type of loop is the for loop. You can use a for loop to create a list of elements in three steps:

1. Instantiate an empty list.
2. Loop over an iterable or [range](https://realpython.com/python-range/) of elements.
3. [Append](https://realpython.com/python-append/) each element to the end of the list.

-- If you want to create a list containing the first ten perfect squares, then you can complete these steps in three lines of code:

squares = []

*for* i in range(10):

    squares.append(i \* i)

print(squares)

-- Here, you instantiate an empty list, squares. Then, you use a for loop to iterate over range(10). Finally, you multiply each number by itself and append the result to the end of the list.

2. Using map() Object:

-- map() provides an alternative approach that’s based in functional programming.

-- You pass in a function and an iterable, and map() will create an object.

-- This object contains the output you would get from running each iterable element through the supplied function.

-- As an example, consider a situation in which you need to calculate the price after tax for a list of transactions:

txns = [1.09, 23.56, 57.84, 4.56, 6.78]

TAX\_RATE = .08

def *get\_price\_with\_tax*(txn):

*return* txn \* (1 + TAX\_RATE)

final\_prices = map(get\_price\_with\_tax, txns)

print(list(final\_prices))

-- Here, you have an iterable txns and a function get\_price\_with\_tax(). You pass both of these arguments to map(), and store the resulting object in final\_prices. You can easily convert this map object into a list using list().

3. Using List Comprehension:

-- List comprehensions are a third way of making lists.

-- With this elegant approach, you could rewrite the for loop from the first example in just a single line of code:

squares = [i \* i *for* i in range(10)]

print(squares)

-- Rather than creating an empty list and adding each element to the end, you simply define the list and its contents at the same time by following this format:

new\_list = [expression for member in iterable]

-- Every list comprehension in Python includes three elements:

1. **expression** is the member itself, a call to a method, or any other valid expression that returns a value. In the example above, the expression i \* i is the square of the member value.
2. **member** is the object or value in the list or iterable. In the example above, the member value is i.
3. **iterable** is a list, set, sequence, [generator](https://realpython.com/introduction-to-python-generators/), or any other object that can return its elements one at a time. In the example above, the iterable is range(10).

-- Because the **expression** requirement is so flexible, a list comprehension in Python works well in many places where you would use map().

-- You can rewrite the pricing example with its own list comprehension:

txns = [1.09, 23.56, 57.84, 4.56, 6.78]

TAX\_RATE = .08

def *get\_price\_with\_tax*(txn):

*return* txn \* (1 + TAX\_RATE)

final\_prices = [get\_price\_with\_tax(i) *for* i in txns]

print(final\_prices)

-- The only distinction between this implementation and map() is that the list comprehension in Python returns a list, not a map object.

**-- Benefits of Using List Comprehension**

-- List comprehensions are often described as being more Pythonic than loops or map().

-- One main benefit of using a list comprehension in Python is that it’s a single tool that you can use in many different situations.

-- In addition to standard list creation, list comprehensions can also be used for mapping and filtering. You don’t have to use a different approach for each scenario.

-- List comprehensions are also more **declarative** than loops, which means they’re easier to read and understand.

-- Loops require you to focus on how the list is created.

-- You have to manually create an empty list, loop over the elements, and add each of them to the end of the list.

-- With a list comprehension in Python, you can instead focus on what you want to go in the list and trust that Python will take care of how the list construction takes place.

**-- How To Supercharge Your Comprehensions**

-- In order to understand the full value that list comprehensions can provide, it’s helpful to understand their range of possible functionality.

1. Using Conditional Logic

-- Earlier, you saw this formula for how to create list comprehensions:

new\_list = [expression for member in iterable]

-- While this formula is accurate, it’s also a bit incomplete. A more complete description of the comprehension formula adds support for optional **conditionals**.

-- The most common way to add conditional logic to a list comprehension is to add a conditional to the end of the expression:

new\_list = [expression for member in iterable (if conditional)]

-- Here, your conditional statement comes just before the closing bracket.

-- Conditionals are important because they allow list comprehensions to filter out unwanted values, which would normally require a call to filter():

sentence = "India's Chandrayan-3 mission is a huge success!"

vowels = [i *for* i in sentence *if* i in 'aeiou']

print(vowels)

-- In this code block, the conditional statement filters out any characters in sentence that aren’t a vowel.

-- The conditional can test any valid expression. If you need a more complex filter, then you can even move the conditional logic to a separate function:

sentence = 'The rocket, who was named Chandrayan-3, came back from Moon because he missed his friends.'

def *is\_consonant*(letter):

    vowels = 'aeiou'

*return* letter.isalpha() and letter.lower() not in vowels

consonants = [i *for* i in sentence *if* is\_consonant(i)]

print(consonants)

-- Here, you create a complex filter is\_consonant() and pass this function as the conditional statement for your list comprehension.

-- Note that the member value i is also passed as an argument to your function.

1. append()

-- The append() method adds an item to the end of the list.

-- The syntax of the append() method is:

list.append(item)

currencies = ['Dollar', 'Euro', 'Pound']

# *append 'Rupee' to the list*

currencies.append('Rupee')

print(currencies)

-- Appending a list to the existing list:

# *Adding a list to the list*

# *animals list*

animals = ['cat', 'dog', 'rabbit']

# *list of wild animals*

wild\_animals = ['tiger', 'fox']

# *appending wild\_animals list to animals*

animals.append(wild\_animals)

print('Updated animals list: ', animals)

2. extend()

-- The extend() method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list.

-- The syntax of the extend() method is:

list1.extend(iterable)

Here, all the elements of iterable are added to the end of list1.

-- The extend() method modifies the original list. It doesn't return any value.

# *create a list*

prime\_numbers = [2, 3, 5]

# *create another list*

numbers = [1, 4]

# *add all elements of prime\_numbers to numbers*

numbers.extend(prime\_numbers)

print('List after extend():', numbers)

# *o/p : List after extend(): [1, 4, 2, 3, 5]*

-- Add elements of tuple and set to the list:

# *Adding elements of tuple and set to the list*

# *languages list*

languages = ['French']

# *languages tuple*

languages\_tuple = ('Spanish', 'Portuguese')

# *languages set*

languages\_set = {'Chinese', 'Japanese'}

# *appending language\_tuple elements to language*

languages.extend(languages\_tuple)

print('New Language List:', languages)

# *appending language\_set elements to language*

languages.extend(languages\_set)

print('Newer Languages List:', languages)

-- Other ways to extend list:

-- You can also append all elements of an iterable to the list using:

**1. the + operator:**

# *Using + operator*

a = [1, 2]

b = [3, 4]

a += b    # *a = a + b*

print('a =', a)

# *Output: [1, 2, 3, 4]*

**2. the list slicing:**

# *Using list slicing*

a = [1, 2]

b = [3, 4]

a[len(a):] = b

print('a =', a)

# *Output: [1, 2, 3, 4]*

3. insert()

-- The insert() method inserts an element to the list at the specified index.

-- The syntax of the insert() method is

list.insert(i, elem)

Here, elem is inserted to the list at the ith index. All the elements after elem are shifted to the right.

-- The insert() method doesn't return anything; returns None. It only updates the current list.

# *create a list of prime numbers*

prime\_numbers = [2, 3, 5, 7]

# *insert 11 at index 4*

prime\_numbers.insert(4, 11)

print('List:', prime\_numbers)

-- Inserting a tuple (as an element) to a list

mixed\_list = [{1, 2}, [5, 6, 7]]

# *number tuple*

number\_tuple = (3, 4)

# *inserting a tuple to the list*

mixed\_list.insert(1, number\_tuple)

print('Updated List:', mixed\_list)

# *o/p: Updated List: [{1, 2}, (3, 4), [5, 6, 7]]*

4. remove()

-- The remove() method removes the first matching element (which is passed as an argument) from the list.

-- The syntax of the remove() method is:

list.remove(element)

-- The remove() doesn't return any value (returns None).

# *animals list*

animals = ['cat', 'dog', 'rabbit', 'guinea pig']

# *'rabbit' is removed*

animals.remove('rabbit')

# *Updated animals List*

print('Updated animals list: ', animals)

# *Output: Updated animals list:  ['cat', 'dog', 'guinea pig']*

-- If a list contains duplicate elements, the remove() method only removes the first matching element.

-- Here, only the first occurrence of element 'dog' is removed from the list.

# *animals list*

animals = ['cat', 'dog', 'dog', 'guinea pig', 'dog']

# *'dog' is removed*

animals.remove('dog')

# *Updated animals list*

print('Updated animals list: ', animals)

# *Output: Updated animals list:  ['cat', 'dog', 'guinea pig', 'dog']*

-- Deleting an element that does not exist in the list:

# *animals list*

animals = ['cat', 'dog', 'rabbit', 'guinea pig']

# *Deleting 'fish' element*

animals.remove('fish')

# *Updated animals List*

print('Updated animals list: ', animals)

# o/p: Traceback (most recent call last):

File "g:\Python Tutorial\Beginners Tutorial\Tutorial\_1\list\_methods.py", line 103, in <module>

animals.remove('fish')

ValueError: list.remove(x): x not in list

-- Here, we are getting an error because the animals list doesn't contain 'fish'.

5. count()

-- The count() method returns the number of times the specified element appears in the list.

-- The syntax of the count() method is:

list.count(element)

-- The count() method returns the number of times element appears in the list.

# *vowels list*

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

# *count element 'i'*

count = vowels.count('i')

# *print count*

print('The count of i is:', count)

# *o/p: The count of i is: 2*

# *count element 'p'*

count = vowels.count('p')

# *print count*

print('The count of p is:', count)

# *o/p: The count of i is: 0*

-- To count tuple and list elements inside a list:

# *random list*

random = ['a', ('a', 'b'), ('a', 'b'), [3, 4]]

# *count element ('a', 'b')*

count = random.count(('a', 'b'))

# *print count*

print("The count of ('a', 'b') is:", count)

# *o/p: The count of ('a', 'b') is: 2*

# *count element [3, 4]*

count = random.count([3, 4])

# *print count*

print("The count of [3, 4] is:", count)

# *o/p: The count of [3, 4] is: 1*

6. pop()

-- The list pop() method removes the item at the specified index. The method also returns the removed item.

-- The syntax of the pop() method is:

list.pop(index)

-- If the index passed to the method is not in range, it throws **IndexError: pop index out of range** exception.

-- The pop() method returns the item present at the given index. This item is also removed from the list.

# *programming languages list*

languages = ['Python', 'Java', 'C++', 'French', 'C']

# *remove and return the 4th item*

return\_value = languages.pop(3)

print('Return Value:', return\_value)

# *o/p: Return Value: French*

# *Updated List*

print('Updated List:', languages)

# *o/p: Updated List: ['Python', 'Java', 'C++', 'C']*

-- pop() without an index and for negative index:

# *programming languages list*

languages = ['Python', 'Java', 'C++', 'Ruby', 'C']

# *remove and return the last item*

print('When index is not passed:')

print('Return Value:', languages.pop())

print('Updated List:', languages)

# *o/p: When index is not passed:*

# *Return Value: C*

# *Updated List: ['Python', 'Java', 'C++', 'Ruby']*

# *remove and return the last item*

print('\nWhen -1 is passed:')

print('Return Value:', languages.pop(-1))

print('Updated List:', languages)

# *o/p: When -1 is passed:*

# *Return Value: Ruby*

# *Updated List: ['Python', 'Java', 'C++']*

# *remove and return the third last item*

print('\nWhen -3 is passed:')

print('Return Value:', languages.pop(-3))

print('Updated List:', languages)

# *o/p: When -3 is passed:*

# *Return Value: Python*

# *Updated List: ['Java', 'C++']*

7. reverse()

-- The reverse() method reverses the elements of the list.

-- The syntax of the reverse() method is:

list.reverse()

-- The reverse() method doesn't return any value. It updates the existing list.

# *Operating System List*

systems = ['Windows', 'macOS', 'Linux']

print('Original List:', systems)

# *List Reverse*

systems.reverse()

# *updated list*

print('Updated List:', systems)

-- Reverse a list using slicing operator:

# *Operating System List*

systems = ['Windows', 'macOS', 'Linux']

print('Original List:', systems)

# *Reversing a list*

# *Syntax: reversed\_list = systems[start:stop:step]*

reversed\_list = systems[::-1]

# *updated list*

print('Updated List:', reversed\_list)

-- Accessing elements in reversed order: If you need to access individual elements of a list in the reverse order, it's better to use the reversed() function.

# *Operating System List*

systems = ['Windows', 'macOS', 'Linux']

# *Printing Elements in Reversed Order*

*for* o in reversed(systems):

    print(o)

8. sort()

-- The sort() method sorts the items of a list in ascending or descending order.

-- The syntax of the sort() method is:

list.sort(key=..., reverse=...)

-- Alternatively, you can also use Python's built-in sorted() function for the same purpose.

sorted(list, key=..., reverse=...)

-- The simplest difference between sort() and sorted() is: sort() changes the list directly and doesn't return any value, while sorted() doesn't change the list and returns the sorted list.

-- By default, sort() doesn't require any extra parameters. However, it has two optional parameters:

* **reverse** - If True, the sorted list is reversed (or sorted in Descending order)
* **key** - function that serves as a key for the sort comparison

-- The sort() method doesn't return any value. Rather, it changes the original list.

-- If you want a function to return the sorted list rather than change the original list, use sorted().

*# vowels list*

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

# *sort the vowels*

vowels.sort()

# *print vowels*

print('Sorted list:', vowels)

# *o/p: Sorted list: ['a', 'e', 'i', 'o', 'u']*

-- Sort in descending order:

-- The sort() method accepts a reverse parameter as an optional argument.

-- Setting reverse = True sorts the list in the descending order.

# *vowels list*

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

# *sort the vowels*

vowels.sort(reverse=True)

# *print vowels*

print('Sorted list (in Descending):', vowels)

# *o/p: Sorted list (in Descending): ['u', 'o', 'i', 'e', 'a']*