

Questions to Test Your Knowledge of Python Lists

1. Check if a list contains an element

The `in` operator will return True if a specific element is in a list.

```
li = [1,2,3,'a','b','c'] 'a' in li    #=> True
```

2. How to iterate over 2+ lists at the same time

You can `zip()` lists and then iterate over the `zip` object. A `zip` object is an iterator of tuples.

Below we iterate over 3 lists simultaneously and interpolate the values into a string.

```
name = ['Snowball', 'Chewy', 'Bubbles', 'Gruff']
animal = ['Cat', 'Dog', 'Fish', 'Goat']
age = [1, 2, 2, 6]

z = zip(name, animal, age)

z #=> <zip at 0x111081e48>

for name, animal, age in z:
    print("%s the %s is %s" % (name, animal, age))

#=> Snowball the Cat is 1
#=> Chewy the Dog is 2
#=> Bubbles the Fish is 2
#=> Gruff the Goat is 6
```

3. When would you use a list vs dictionary?

Lists and dictionary generally have slightly different use cases but there is some overlap.

The **general rule of algorithm questions** I've come to is that if you can use both, use a dictionary because lookups are faster.

List

Use a **list** if you need to store the order of something.

Ie: id's of database records in the order they'll be displayed.

```
ids = [23,1,7,9]
```

While both lists and dictionaries are ordered as of python 3.7, a list allows duplicate values while a dictionary doesn't allow duplicate keys.

Dictionary:

Use a dictionary if you want to count occurrences of something. Like the number of pets in a home.

```
pets = {'dogs':2, 'cats':1, 'fish':5}
```

Each key can only exist once in a dictionary. Note that keys can also be other immutable data structures like tuples. Ie: `{('a',1):1, ('b',2):1}`.

4. Is a list mutable?

Yes. Notice in the code below how the value associated with the same identifier in memory has not changed.

```
x = [1]

print(id(x), ': ', x) #=> 4501046920 : [1]

x.append(5)

x.extend([6,7])

print(id(x), ': ', x) #=> 4501046920 : [1, 5, 6, 7]
```

5. Does a list need to be homogeneous?

No. Different types of object can be mixed together in a list.

```
a = [1, 'a', 1.0, []]

a  #=> [1, 'a', 1.0, []]
```

6. What is the difference between append and extend?

`.append()` adds an object to the end of a list.

```
a = [1, 2, 3]

a.append(4)

a  #=> [1, 2, 3, 4]
```

This also means appending a list adds that whole list as a single element, rather than appending each of its values.

```
a.append([5, 6])

a  #=> [1, 2, 3, 4, [5, 6]]
```

`.extend()` adds each value from a 2nd list as its own element. So extending a list with another list combines their values.

```
b = [1, 2, 3]

b.extend([5, 6])

b  #=> [1, 2, 3, 5, 6]
```

7. Do python lists store values or pointers?

Python lists don't store values themselves. They store pointers to values stored elsewhere in memory. This allows lists to be mutable.

Here we initialize values `1` and `2`, then create a list including the values `1` and `2`.

```
print( id(1) )  #=> 4438537632

print( id(2) )  #=> 4438537664

a = [1, 2, 3]

print( id(a) )  #=> 4579953480

print( id(a[0]) )  #=> 4438537632

print( id(a[1]) )  #=> 4438537664
```

Notice how the list has its own memory address. But `1` and `2` in the list point to the same place in memory as the `1` and `2` we previously defined.

8. What does “del” do?

`del` removes an item from a list given its index.

Here we’ll remove the value at index 1.

```
a = ['w', 'x', 'y', 'z']  
a #=> ['w', 'x', 'y', 'z']  
  
del a[1]  
  
a #=> ['w', 'y', 'z']
```

Notice how `del` does not return the removed element.

9. What is the difference between “remove” and “pop”?

`.remove()` removes the first instance of a matching object. Below we remove the first `b`.

```
a = ['a', 'a', 'b', 'b', 'c', 'c']  
  
a.remove('b')  
  
a #=> ['a', 'a', 'b', 'c', 'c']
```

`.pop()` removes an object by its index.

The difference between `pop` and `del` is that `pop` returns the popped element. This allows using a list like a stack.

```
a = ['a', 'a', 'b', 'b', 'c', 'c']  
a.pop(4) #=> 'c'  
a #=> ['a', 'a', 'b', 'b', 'c']
```

By default, `pop` removes the last element from a list if an index isn’t specified.

10. Remove duplicates from a list

If you're not concerned about maintaining the order of a list, then converting to a set and back to a list will achieve this.

```
li = [3, 2, 2, 1, 1, 1]

list(set(li)) #=> [1, 2, 3]
```

11. Find the index of the 1st matching element

For example, you want to find the first “apple” in a list of fruit. Use the `.index()` method.

```
fruit = ['pear', 'orange', 'apple', 'grapefruit', 'apple', 'pear']

fruit.index('apple') #=> 2

fruit.index('pear') #=> 0
```

12. Remove all elements from a list

Rather than creating a new empty list, we can clear the elements from an existing list with `.clear()`.

```
fruit = ['pear', 'orange', 'apple']
print( fruit )

#=> ['pear', 'orange', 'apple']

print( id(fruit) ) #=> 4581174216

fruit.clear()

print( fruit )      #=> []

print( id(fruit) ) #=> 4581174216
```

Or with `del`.

```
fruit = ['pear', 'orange', 'apple']

print( fruit )
```

```
#=> ['pear', 'orange', 'apple']

print( id(fruit) ) #=> 4581166792

del fruit[:]

print( fruit )      #=> []

print( id(fruit) ) #=> 4581166792
```

13. Iterate over both the values in a list and their indices

`enumerate()` adds a counter to the list passed as an argument.

Below we iterate over the list and pass both value and index into string interpolation.

```
grocery_list = ['flour', 'cheese', 'carrots']
for idx , val in enumerate(grocery_list):
    print("%s: %s" % (idx, val))

#=> 0: flour
#=> 1: cheese
#=> 2: carrots
```

14. How to concatenate two lists

The `+` operator will concatenate 2 lists.

```
one = ['a', 'b', 'c']

two = [1, 2, 3]

one + two #=> ['a', 'b', 'c', 1, 2, 3]
```

15. How to manipulate every element in a list with list comprehension

Below we return a new list with 1 added to every element.

```
li = [0,25,50,100]

[i+1 for i in li]

#=> [1, 26, 51, 101]
```

16. Count the occurrence of a specific object in a list

The `count()` method returns the number of occurrences of a specific object. Below we return the number of times the string, "fish" exists in a list called `pets`.

```
pets = ['dog', 'cat', 'fish', 'fish', 'cat']  
  
pets.count('fish')  
  
#=> 2
```

17. How to shallow copy a list?

`.copy()` can be used to shallow copy a list.

Below we create a shallow copy of `round1`, assign it to a new name, `round2`, and then remove the string `sonny chiba`.

For example:

```
round1 = ['chuck norris', 'bruce lee', 'sonny chiba']  
  
round2 = round1.copy()  
  
round2.remove('sonny chiba')  
  
print(round1)  
  
#=> ['chuck norris', 'bruce lee', 'sonny chiba']  
  
print(round2) #=> ['chuck norris', 'bruce lee']
```

18. Why create a shallow copy of a list?

Building off the previous example, modifying `round2` will modify `round1` if we don't create a shallow copy.

```
round1 = ['chuck norris', 'bruce lee', 'sonny chiba']  
  
round2 = round1  
  
round2.remove('sonny chiba')  
  
print(round1) #=> ['chuck norris', 'bruce lee']
```

```
print(round2) #=> ['chuck norris', 'bruce lee']
```

Without a shallow copy, `round1` and `round2` are just names pointing to the same list in memory. That's why it appears that changing the value of one changes the value of the other.

19. How to deep copy a list?

For this we need to import the `copy` module, then call `copy.deepcopy()`.

Below we create a deep copy of a list, `round1` called `round2`, update a value in `round2`, then print both. In this case, `round1` isn't affected.

```
round1 = [
    ['Arnold', 'Sylvester', 'Jean Claude'],
    ['Buttercup', 'Bubbles', 'Blossom']
]

import copy

round2 = copy.deepcopy(round1)

round2[0][0] = 'Jet Lee'

print(round1)
#=> [['Arnold', 'Sylvester', 'Jean Claude'], ['Buttercup', 'Bubbles', 'Blossom']]

print(round2)
#=> [['Jet Lee', 'Sylvester', 'Jean Claude'], ['Buttercup', 'Bubbles', 'Blossom']]
```

Above we can see that changing the nested array in `round2` did not update `round1`.

20. What is the difference between a deep copy and a shallow copy?

Building off the previous example, creating a shallow copy and then modifying it would have affected the original list..

```
round1 = [
    ['Arnold', 'Sylvester', 'Jean Claude'],
    ['Buttercup', 'Bubbles', 'Blossom']
]
```



```
import copy

round2 = round1.copy()

round2[0][0] = 'Jet Lee'

print(round1)
#=> [['Jet Lee', 'Sylvester', 'Jean Claude'], ['Buttercup', 'Bubbles',
'Blossom']]

print(round2)
#=> [['Jet Lee', 'Sylvester', 'Jean Claude'], ['Buttercup', 'Bubbles',
'Blossom']]
```

Why does this happen?

Creating a shallow copy does create a new object in memory, but it's filled with the same references to existing objects that the previous list has.

Creating a deep copy creates copies of the original objects and points to these new versions. So the new list is completely unaffected by changes to the old list and vice versa.

21. What is the difference between a list and a tuple.

Tuples cannot be updated after creation. Adding/removing/updating an existing tuple requires creating a new tuple.

Lists can be modified after creation.

Tuples often represent an object like a record loaded from a database where elements are of different datatypes.

Lists are generally used to store an ordered sequence of a specific type of object (but not always).

Both are sequences and allow duplicate values.

22. Return the length of a list

`len()` can returns the length of a list.

```
li = ['a', 'b', 'c', 'd', 'e']  
  
len(li)  
  
#=> 5
```

But note it counts top level objects, so a nested list of several integers will only be counted as a single object. Below, `li` has a length of 2, not 5.

```
li = [[1,2],[3,4,5]]  
  
len(li)  
  
#=> 2
```

23. What is the difference between a list and a set?

While a list is ordered, a set is not. That's why using set to find unique values in a list, like `list(set([3, 3, 2, 1]))` loses the order.

While lists are often used to track order, sets are often used to track existence.

Lists allow duplicates, but all values in a set are unique by definition.

24. How to check if an element is not in a list?

For this we use the `in` operator, but prefix it with `not`.

```
li = [1,2,3,4]  
  
5 not in li #=> True  
  
4 not in li #=> False
```

25. Multiply every element in a list by 5 with the map function

`.map()` allows iterating over a sequence and updating each value with another function.

`map()` returns a map object but I've wrapped it with a list comprehension so we can see the updated values.

```
def multiply_5(val):  
    return val * 5
```

25. Multiply every element in a list by 5 with the map function

`.map()` allows iterating over a sequence and updating each value with another function.

`map()` returns a map object but I've wrapped it with a list comprehension so we can see the updated values.

```
def multiply_5(val):  
    return val * 5  
  
a = [10, 20, 30, 40, 50]  
  
[val for val in map(multiply_5, a)]  
  
#=> [50, 100, 150, 200, 250]
```

26. Combine 2 lists into a list of tuples with the zip function

`zip()` combines multiple sequences into an iterator of tuples, where values at the same sequence index are combined in the same tuple.

```
alphabet = ['a', 'b', 'c']  
integers = [1, 2, 3]  
  
list(zip(alphabet, integers))
```

27. Insert a value at a specific index in an existing list

The `insert()` method takes an object to insert and the index to insert it at.

```
li = ['a', 'b', 'c', 'd', 'e']  
  
li.insert(2, 'HERE')li #=> ['a', 'b', 'HERE', 'c', 'd', 'e']
```

Note that the element previously at the specified index is shifted to the right, not overwritten.

28. Subtract values in a list from the first element with the reduce function

`reduce()` needs to be imported from `functools`.

Given a function, `reduce` iterates over a sequence and calls the function on every element. The output from the previous element is passed as an argument when calling the function on the next element.

```
from functools import reduce

def subtract(a,b):
    return a - b

numbers = [100,10,5,1,2,7,5]

reduce(subtract, numbers) #=> 70
```

Above we subtracted 10, 5, 1, 2, 7 and 5 from 100.

29. Remove negative values from a list with the filter function

Given a function, `filter()` will remove any elements from a sequence on which the function doesn't return `True`.

Below we remove elements less than zero.

```
def remove_negatives(x):
    return True if x >= 0 else False

a = [-10, 27, 1000, -1, 0, -30]

[x for x in filter(remove_negatives, a)]

#=> [27, 1000, 0]
```

30. Convert a list into a dictionary where list elements are keys

For this we can use a dictionary comprehension.

```
li = ['The', 'quick', 'brown', 'fox', 'was', 'quick']  
d = {k:1 for k in li}  
d #=> {'The': 1, 'quick': 1, 'brown': 1, 'fox': 1, 'was': 1}
```

31. Modify an existing list with a lambda function

Let's take the previous `map` function we wrote and turn it into a one-liner with a `lambda`.

```
a = [10,20,30,40,50]  
list(map(lambda val:val*5, a))  
#=> [50, 100, 150, 200, 250]
```

I could have left it as a map object until I needed to iterate over it but I converted to a list to show the elements inside.

32. Remove elements in a list after a specific index

Using the slice syntax, we can return a new list with only the elements up to a specific index.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]  
li[:10]  
#=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

33. Remove elements in a list before a specific index

The slice syntax can also return a new list with the values after a specified index.

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

```
li[15:]  
  
#=> [16, 17, 18, 19, 10]
```

34. Remove elements in a list between 2 indices

Or between two indices.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]  
  
li[12:17]  
  
#=> [13, 14, 15, 16, 17]
```

35. Return every 2nd element in a list between 2 indices

Or before/after/between indices at a specific interval.

Here we return every 2nd value between the indices 10 and 16 using the slice syntax.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]  
  
li[10:16:2]  
  
#=> [11, 13, 15]
```

36. Sort a list of integers in ascending order

The `sort()` method mutates a list into ascending order.

```
li = [10,1,9,2,8,3,7,4,6,5]li.sort()  
  
li #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

37. Sort a list of integers in descending order

It's also possible to sort in descending order with `sort()` by adding the argument `reverse=True`.

```
li = [10,1,9,2,8,3,7,4,6,5]

li.sort(reverse=True)

li #=> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

38. Filter even values out of a list with list comprehension

You can add conditional logic inside a list comprehension to filter out values following a given pattern.

Here we filter out values divisible by 2.

```
li = [1,2,3,4,5,6,7,8,9,10]

[i for i in li if i % 2 != 0]

#=> [1, 3, 5, 7, 9]
```

39. Count occurrences of each value in a list

One option is to iterate over a list and add counts to a dictionary. But the easiest option is to import the `Counter` class from `collections` and pass the list to it.

```
from collections import Counter

li = ['blue', 'pink', 'green', 'green', 'yellow', 'pink', 'orange']
Counter(li)

#=> Counter({'blue': 1, 'pink': 2, 'green': 2, 'yellow': 1, 'orange': 1})
```

40. Get the first element from each nested list in a list

A list comprehension is well suited for iterating over a list of other objects and grabbing an element from each nested object.

```
li = [[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13,14,15]]

[i[0] for i in li]  #=> [1, 4, 7, 10, 13]
```

41. What is the time complexity of insert, find and delete for a list?

Insert is $O(n)$. If an element is inserted at the beginning, all other elements must be shifted right.

Find by index is $O(1)$. But find by value is $O(n)$ because elements need to be iterated over until the value is found.

Delete is $O(n)$. If an element is deleted at the beginning, all other elements must be shifted left.

42. Combine elements in a list into a single string.

This can be done with the `join()` function.

```
li = ['The','quick','brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']  
  
' '.join(li)  
  
#=> 'The quick brown fox jumped over the lazy dog'
```

43. What's the affect of multiplying a list by an integer?

Multiplying a list by an integer is called multiple concatenation and has the same affect as concatenating a list to itself n-times.

Below we multiply a list by 5.

```
['a','b'] * 5  
  
#=> ['a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'a', 'b']
```

This is the same as.

```
['a','b'] + ['a','b'] + ['a','b'] + ['a','b'] + ['a','b']  
  
#=> ['a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'a', 'b']
```


44. Use the “any” function to return True if any value in a list is divisible by 2

We can combine `any()` with a list comprehension to return `True` if any values in the returned list evaluate to `True`.

Below the 1st list comprehension returns `True` because the list has a `2` in it, which is divisible by `2`.

```
li1 = [1,2,3]
li2 = [1,3]
any(i % 2 == 0 for i in li1) #=> True
any(i % 2 == 0 for i in li2) #=> False
```

45. Use the “all” function to return True if all values in a list are negative

Similar to the `any()` function, `all()` can also be used with a list comprehension to return `True` only if all values in the returned list are `True`.

```
li1 = [2,3,4]
li2 = [2,4]
all(i % 2 == 0 for i in li1) #=> False
all(i % 2 == 0 for i in li2) #=> True
```

46. Can you sort a list with “None” in it?

You cannot sort a list with `None` in it because comparison operators (used by `sort()`) can’t compare an integer with `None`.

```
li = [10,1,9,2,8,3,7,4,6,None]
li.sort()
li #=> TypeError: '<' not supported between instances of 'NoneType' and 'int'
```

47. What kind of copy would the list constructor create from an existing list?

The list constructor creates a shallow copy of a passed in list. That said, this is less pythonic than using `.copy()`.

```
li1 = ['a','b']  
li2 = list(li1)  
li2.append('c')  
print(li1) #=> ['a', 'b']  
print(li2) #=> ['a', 'b', 'c']
```

48. Reverse the order of a list

A list can be mutated into reverse order with the `reverse()` method.

```
li = [1,2,3,4,5,6,7,8,9,10]  
li.reverse()  
li #=> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note that this mutates the object instead of returning a new object.

49. What is the difference between reverse and reversed?

`reverse()` reverses the list in place. `reversed()` returns an iterable of the list in reverse order.

```
li = [1,2,3,4,5,6,7,8,9,10]  
list(reversed(li))  
#=> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

50. What is the difference between sort and sorted?

`sort()` modifies the list in place. `sorted()` returns a new list in reverse order.

```
li = [10,1,9,2,8,3,7,4,6,5]
```

```
li.sort()

li #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

li = [10,1,9,2,8,3,7,4,6,5]

sorted(li) #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

51. Return the minimum value in a list

The `min()` function returns the minimum value in a list.

```
li = [10,1,9,2,8,3,7,4,6,5]

min(li)  #=> 1
```

52. Return the maximum value in a list

The `max()` function returns the maximum value in a list.

```
li = [10,1,9,2,8,3,7,4,6,5]

max(li)  #=> 10
```

53. Return the sum of values in a list

The `sum()` function returns the sum of all values in a list.

```
li = [10,1,9,2,8,3,7,4,6,5]

sum(li)  #=> 55
```

54. Use a list as a stack

You can use `append()` and `pop()` to treat a list like a stack. Stacks function per LIFO (last in first out).

```
stack = []

stack.append('Jess')

stack.append('Todd')

stack.append('Yuan')
```

```
print(stack) #=> ['Jess', 'Todd', 'Yuan']  
print(stack.pop()) #=> Yuan  
print(stack) #=> ['Jess', 'Todd']
```

One benefit of a stack is that elements can be added and removed in $O(1)$ time because the list does not need to be iterated over.

55. Find the intersection of 2 lists

We can do this by utilizing `set()` with an ampersand.

```
li1 = [1,2,3]  
li2 = [2,3,4]  
set(li1) & set(li2)  #=> {2, 3}
```

56. Find the difference between a set and another set

We can't subtract lists, but we can subtract sets.

```
li1 = [1,2,3]  
li2 = [2,3,4]  
set(li1) - set(li2)  #=> {1}  
set(li2) - set(li1)  #=> {4}
```

57. Flatten a list of lists with a list comprehensions

Unlike Ruby, Python3 doesn't have an explicit flatten function. But we can use list comprehension to flatten a list of lists.

```
li = [[1,2,3], [4,5,6]]  
[i for x in li for i in x]  #=> [1, 2, 3, 4, 5, 6]
```

58. Generate a list of every integer between 2 values

We can create a range between 2 values and then convert that to a list.

```
list(range(5,10))    #=> [5, 6, 7, 8, 9]
```

59. Combine 2 lists into a dictionary

Using `zip()` and the `list()` constructor we can combine 2 lists into a dictionary where one list becomes the keys and the other list becomes the values.

```
name = ['Snowball', 'Chewy', 'Bubbles', 'Gruff']
animal = ['Cat', 'Dog', 'Fish', 'Goat']dict(zip(name, animal))
#=> {'Snowball': 'Cat', 'Chewy': 'Dog', 'Bubbles': 'Fish', 'Gruff': 'Goat'}
```

60. Reverse the order of a list using the slice syntax

While we can reverse a list with `reverse()` and `reversed()`, it can also be done with the slice syntax.

This returns a new list by iterating back over the list from end to beginning.

```
li = ['a','b',3,4]
li[::-1]    #=> [4, 3, 'b', 'a']
```