Q1. Can you create a programme or function that employs both positive and negative indexing? Is there any repercussion if you do so?

In Python, you can start indexing from the end of an iterable. This is known as negative indexing.

last = list\_items[-1]

For example, let’s get the last value of a list:

numbers = [1, 2, 3, 4, 5]

last = numbers[-1]

print(last)

Output:

5

Here is an illustration of how the list indexing works in Python:

This means you can use both positive and negative indexes to access iterables.

In this guide, you learn about negative indexing and its applications in Python.

This means you can use both positive and negative indexes to access iterables.

In this guide, you learn about negative indexing and its applications in Python.

Indexing in Python

To access elements of a Python iterable, such as a list, you need to know the index of the element.

Python supports indexing in two ways:

Positive zero-based indexing.

Negative indexing that “starts from the end”.

Let’s take a closer look at both of these.

Zero-Based Indexing in Python

The basic way to access iterable elements in Python is by using positive zero-based indexing.

This means each element in the iterable can be referred to with an index starting from 0.

In zero-based indexing, the 1st element has 0 index, the 2nd element has 1, and so on. Here is an illustration:

For example, let’s get the first name in a list of names:

names = ["Alice", "Bob", "Charlie", "David", "Emmanuel", "Fiona"]

first = names[0]

print(first)

Output:

Python

Negative “From the End” Indexing in Python

Python supports “indexing from the end”, that is, negative indexing.

This means the last value of a sequence has an index of -1, the second last -2, and so on.

Negative “From the End” Indexing in Python

Python supports “indexing from the end”, that is, negative indexing.

This means the last value of a sequence has an index of -1, the second last -2, and so on.

For instance, let’s get the last three names from a list of names:

names = ["Alice", "Bob", "Charlie", "David", "Emmanuel", "Fiona"]

last = names[-1]

second\_last = names[-2]

third\_last = names[-3]

print(last, second\_last, third\_last)

Output:

Fiona Emmanuel David

Slicing and Indexing in Python

Negative indexing is really useful with slicing iterables in Python.

Slicing in a Nutshell

In short, slicing means retrieving the desired subsequence of a sequence in Python. Slicing iterable returns a [slice](https://docs.python.org/3/c-api/slice.html) object. This is a part of the iterable, such as the first three numbers of a list of ten numbers.

Slicing follows this syntax:

iterable[start:stop:step]

Where:

start is the starting index of the slice.

stop determines the end of the slice. The stop index is excluded from the slice!

step determines the step size of how many elements to jump over when slicing.

For example, let’s get the first three names from a list of names:

names = ["Alice", "Bob", "Charlie", "David", "Emmanuel", "Fiona"]

first\_three = names[0:3:1]

print(first\_three)

Output:

['Alice', 'Bob', 'Charlie']

It is useful to know:

You can leave out the start parameter if you want to start from the beginning of the iterable.

If you want to slice until the end of the iterable, you can omit the stop parameter.

And if you want to take steps of size 1, you can leave the step parameter out too.

So the above example becomes:

first\_three = names[:3]

Slicing “From the End” with Negative Indexes

In Python, slicing supports negative indexing too. This makes it easier to slice iterables from the end.

For instance, let’s get the last three names from the list:

names = ["Alice", "Bob", "Charlie", "David", "Emmanuel", "Fiona"]

last\_three = names[-3:]

print(last\_three)

Output:

['David', 'Emmanuel', 'Fiona']

Negative slicing also supports negative step size. This makes the slicing go backward.

For example, let’s reverse the list of names. To do this, you do not need start and stop parameters, as you start from the beginning and stop at the end:

names = ["Alice", "Bob", "Charlie", "David", "Emmanuel", "Fiona"]

last\_three = names[::-1]

print(last\_three)

Output:

['Fiona', 'Emmanuel', 'David', 'Charlie', 'Bob', 'Alice']

Conclusion

Today you learned how to start indexing from the end of an iterable in Python.

To recap, Python supports positive zero-based indexing and negative indexing that starts from -1.

Negative indexing in Python means the indexing starts from the end of the iterable. The last element is at index -1, the second last at -2, and so on.

Q2. What is the most effective way of starting with 1,000 elements in a Python list? Assume that all elements should be set to the same value.

The following are some of the ways to initialize lists(we create lists of size 1000 and initialize with zeros) in Python. We create an empty an list and run a for loop for n times using the append() method to add elements to the list. This is similar to the above method. However we use while loop instead.

Python is a very flexible language where a single task can be performed in a number of ways, for example initializing lists can be performed in many ways. However, there are subtle differences in these seemingly similar methods. Python which is popular for its simplicity and readability is equally infamous for being slow compared to C++ or Java. The ‘for’ loop is especially known to be slow whereas methods like map() and filter() are known to be faster because they are written in C. Knowing the better and faster way to initialize lists might give you a slight edge in competitive programming.

The following are some of the ways to initialize lists(we create lists of size 1000 and initialize with zeros) in Python.

Using a for loop and append()  
We create an empty an list and run a for loop for n times using the append() method to add elements to the list.

arr = []

for i in range(1000):

arr.append(0)

Using a while loop with a counter variable  
This is similar to the above method. However we use while loop instead.

arr = []

i = 0

while(i<1000):

arr.append(0)

Using list comprehensions  
It consists of square brackets containing an expression followed by a for clause and further followed by an optional if clause. The expression can be any type of object that we want to put on the list. Since we are initializing the list with zeros, our expression will just be 0.

arr = [0 for i in range(1000)]

Using the \* operator  
The \* operator can be used as [object]\*n where n is the no of elements in the array.

arr = [0]\*1000

Lets look at the time taken by each of them. We will calculate the average time taken by each of these methods to initialize an array of 10000 elements over 500 times.

|  |
| --- |
| # import time module to calculate times  import time    # initialize lists to save the times  forLoopTime = []  whileLoopTime = []  listComprehensionTime = []  starOperatorTime = []    # repeat the process for 500 times  # and calculate average of times taken.  for k in range(500):        # start time      start = time.time()      # declare empty list      a = []      # run a for loop for 10000 times      for i in range(10000):          a.append(0)      # stop time      stop = time.time()      forLoopTime.append(stop-start)        # start time      start = time.time()      # declare an empty list      a = []      i = 0      # run a for loop 10000 times      while(i<10000):          a.append(0)          i+= 1      stop = time.time()      whileLoopTime.append(stop-start)        start = time.time()      # list comprehension to initialize list      a = [0 for i in range(10000)]      stop = time.time()      listComprehensionTime.append(stop-start)          start = time.time()      # using the \* operator      a = [0]\*10000      stop = time.time()      starOperatorTime.append(stop-start)        print("Average time taken by for loop: " + str(sum(forLoopTime)/100))  print("Average time taken by while loop: " + str(sum(whileLoopTime)/100))  print("Average time taken by list comprehensions: " + str(sum(listComprehensionTime)/100))  print("Average time taken by \* operator: " + str(sum(starOperatorTime)/100)) |

Output

Average time taken by for loop: 0.012432687282562256

Average time taken by while loop: 0.017907898426055908

Average time taken by list comprehensions: 0.0034629487991333007

Average time taken by \* operator: 0.0001951146125793457

Note: The times will vary based on the platform where this code is executed. These times are just for studying the relative performance of these methods of initializing.

As it can be seen, for and while loops take almost the same time with for loop having a slight edge.

List comprehensions perform much better than for and while loops with the former being around 3-5 times faster. Another example of this difference can be seen when we try to create a list of numbers from 1-1000. Using list comprehensions is much better than using append().

a = [i for i in range(1, 1001)]

Using the \* operator is way faster than the remaining methods and this is the way you should be initializing lists

However one drawback of using the \* operator is while declaring 2d arrays. Using this operator would create shallow lists i.e only one list object would be created and all the indices would refer to this object. This might create unwanted complications. Hence using list comprehensions is a safer way to create 2d lists.

Using \* operator would create shallow lists

arr = [[0]\*no\_of\_cols]\*no\_of\_rows

Using list comprehensions is better for 2d arrays

arr = [[0 for i in range(no\_of\_cols)] for j in range(no\_of\_rows)]

Q3. How do you slice a list to get any other part while missing the rest? (For example, suppose you want to make a new list with the elements first, third, fifth, seventh, and so on.)

Python Program to Slice Lists

In this example, you will understand different ways of list slicing in Python.

To understand this example, you should have the knowledge of the following [Python programming](https://www.programiz.com/python-programming) topics:

[Python List](https://www.programiz.com/python-programming/list)

The format for list slicing is [start:stop:step].

start is the index of the list where slicing starts.

stop is the index of the list where slicing ends.

step allows you to select nth item within the range start to stop.

List slicing works similar to [Python slice() function](https://www.programiz.com/python-programming/methods/built-in/slice).

Get all the Items

my\_list = [1, 2, 3, 4, 5]

print(my\_list[:])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[1, 2, 3, 4, 5]

If you simply use :, you will get all the elements of the list. This is similar to print(my\_list).

Get all the Items After a Specific Position

my\_list = [1, 2, 3, 4, 5]

print(my\_list[2:])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[3, 4, 5]

If you want to get all the elements after a specific index, you can mention that index before : as shown in example above.

In the above example, elements at index 2 and all the elements after index 2 are printed.

Note: indexing starts from 0. Item on index 2 is also included.

Get all the Items Before a Specific Position

my\_list = [1, 2, 3, 4, 5]

print(my\_list[:2])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[1, 2]

This example lets you get all the elements before a specific index. Mention that index after :.

In the example, the items before index 2 are sliced. Item on index 2 is excluded.

Get all the Items from One Position to Another Position

my\_list = [1, 2, 3, 4, 5]

print(my\_list[2:4])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[3, 4]

If you want to get all the elements between two specific indices, you can mention them before and after :.

In the above example, my\_list[2:4] gives the elements between 2nd and the 4th positions. The starting position (i.e. 2) is included and the ending position (i.e. 4) is excluded.

Get the Items at Specified Intervals

my\_list = [1, 2, 3, 4, 5]

print(my\_list[::2])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[1, 3, 5]

If you want to get elements at specified intervals, you can do it by using two :.

In the above example, the items at interval 2 starting from index 0 are sliced.

If you want the indexing to start from the last item, you can use negative sign -.

my\_list = [1, 2, 3, 4, 5]

print(my\_list[::-2])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[5, 3, 1]

The items at interval 2 starting from the last index are sliced.

If you want the items from one position to another, you can mention them from start to stop.

my\_list = [1, 2, 3, 4, 5]

print(my\_list[1:4:2])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

[2, 4]

The items from index 1 to 4 are sliced with intervals of 2.

Q4. Explain the distinctions between indexing and slicing.

“Indexing” means referring to an element of an iterable by its position within the iterable. “Slicing” means getting a subset of elements from an iterable based on their indices. By way of analogy, I was recently summoned to jury duty, and they assigned each potential juror a number.

Indexing

The term "indexing" refers to refers to an element of an iterable based on its position inside the iterable.

The indexing begins from 0. The first element in the sequence is represented by index 0.

Negative indexing begins from -1. The last element in the sequence is represented by index -1.

Each character in a string corresponds to an index number, and each character can be accessed by its index number. There are two ways to access characters in a String

Accessing string characters using positive indexing

Accessing string characters using negative indexing

T U T O T I A L S

Positive Indexing 0 1 2 3 4 5 6 7 8

Negative indexing -9 -8 -7 -6 -5 -4 -3 -2 -1

Accessing string characters using positive indexing

In this case, we pass a Positive index (that we wish to access) in square brackets. The index number sequence begins with index number 0. (represents the first character of a string).

Example

# input string

inputString = "Hello tutorialspoint python"

print("0th index character:", inputString[0])

print("7th index character", inputString[7])

print("12th index character:", inputString[12])

('0th index character:', 'H')

('7th index character', 'u')

('12th index character:', 'a')

Output

0th index character: H

7th index character u

12th index character: a

Accessing string characters using negative indexing

We pass the Negative index (we wish to access) in square brackets in this type of indexing. The index number starts at -1 in this case (that represents the last character of a string).

Example

# input string

inputString = "Hello tutorialspoint python"

print("last index character:", inputString[-1])

print("6th index character from last:", inputString[-6])

('last index character:', 'n')

('6th index character from last:', 'p')

Output

last index character: n

6th index character from last: p

Indexing in List

Example

# input list

inputList =[1, 4, 8, 6, 2]

print("Element at index 2:", inputList[2])

print("last element of an input list:", inputList[-1])

('Element at index 2:', 8)

('last element of an input list:', 2)

Output

Element at index 2: 8

last element of an input list: 2

NOTE

When we attempt to use an index which does no exist or too large, it throws an IndexError

Example

# input list

inputList =[1, 4, 8, 6, 2]

# printing the element at index 10 of a input list

# throws an IndexError as the index 10 doesn't exist in the input list

print("Element at index 10:", inputList[10])

Traceback (most recent call last):

File "main.py", line 5, in <module>

print("Element at index 10:", inputList[10])

IndexError: list index out of range

Output

Traceback (most recent call last):

File "main.py", line 5, in <module>

print("Element at index 10:", inputList[10])

IndexError: list index out of range

Slicing

The term "slicing" refers to obtaining a subset of elements from an iterable based on their indices.

We create a substring by slicing a string, which is effectively a string that exists within another string. We utilize slicing when we only need a portion of the string and not the entire string.

Syntax

string[start : end : step]

Parameters

start - index from where to start

end - ending index

step - numbers of jumps/increment to take between i.e stepsize

Slicing in strings

# input string

inputString = "Hello tutorialspoint python"

print("First 4 characters of the string:", inputString[: 4])

print("Alternate characters from 1 to 10 index(excluded):", inputString[1 : 10 : 2])

print("Alternate characters in reverse order from 1 to 10 index(excluded):", inputString[-1 : -10 : -2])

('First 4 characters of the string:', 'Hell')

('Alternate characters from 1 to 10 index(excluded):', 'el uo')

('Alternate characters in reverse order from 1 to 10 index(excluded):', 'nhy n')

Output

First 4 characters of the string: Hell

Alternate characters from 1 to 10 index(excluded): el uo

Alternate characters in reverse order from 1 to 10 index(excluded): nhy n

Tuple Slicing

We can use tuple slicing. It is similar to how we use strings and lists. Tuple slicing is used to obtain a variety of items. We also use the slicing operator to perform tuple slicing. The slicing operator can be represented by the syntax

Syntax

[start:stop:step]

Example

# Input tuple

givenTuple = ("Welcome", "this", "is", "TutorialsPoint", "Website", 10)

# Slicing with start and stop values(indices)

print('Tuple slicing from index 1 to index 6 :', givenTuple[1:6])

# Slicing with only stop values(indices)

print("Tuple slicing till index 7: ", givenTuple[:7])

# Slicing with only start value(indices)

print("Tuple slicing from index 2 is:", givenTuple[2:])

# Slicing without any start and stop values

print("Tuple slicing without any start and stop values:", givenTuple[:])

# Slicing in reverse order

print("Tuple slicing in reverse order:", givenTuple[::-1])

('Tuple slicing from index 1 to index 6 :', ('this', 'is', 'TutorialsPoint', 'Website', 10))

('Tuple slicing till index 7: ', ('Welcome', 'this', 'is', 'TutorialsPoint', 'Website', 10))

('Tuple slicing from index 2 is:', ('is', 'TutorialsPoint', 'Website', 10))

('Tuple slicing without any start and stop values:', ('Welcome', 'this', 'is', 'TutorialsPoint', 'Website', 10))

('Tuple slicing in reverse order:', (10, 'Website', 'TutorialsPoint', 'is', 'this', 'Welcome'))

Output

Tuple slicing from index 1 to index 6 : ('this', 'is', 'TutorialsPoint', 'Website', 10)

Tuple slicing till index 7: ('Welcome', 'this', 'is', 'TutorialsPoint', 'Website', 10)

Tuple slicing from index 2 is: ('is', 'TutorialsPoint', 'Website', 10)

Tuple slicing without any start and stop values: ('Welcome', 'this', 'is', 'TutorialsPoint', 'Website', 10)

Tuple slicing in reverse order: (10, 'Website', 'TutorialsPoint', 'is', 'this', 'Welcome')

Indexing vs Slicing Differentiation

The following table shows the key differences between indexing and slicing in python −

|  |  |
| --- | --- |
| Indexing | Slicing |
| It returns only 1 item | It returns a new list/tuple |
| An IndexError will be thrown if you attempt to use an index that is too large. | When used for slicing, out-of-range indexes are handled gently. |
| We cannot change the length of the list by item assignment in indexing. | We can change the length of the list or even clear it by assigning items to slicing. |
| We can assign a single element or an iterable to indexing. | When we assign a single element to slicing, we get a TypeError. It will only accept iterables. |

Q5. What happens if one of the slicing expression's indexes is out of range?

Python List Index Out of Range

If you are working with lists in Python, you have to know the index of the list elements. This will help you access them and perform operations on them such as printing them or looping through the elements. But in case you mention an index in your code that is outside the range of the list, you will encounter an IndexError.

“List index out of range” error occurs in Python when we try to access an undefined element from the list.

The only way to avoid this error is to mention the indexes of list elements properly.

Example:

# Declaring list

list\_fruits = ['apple', 'banana', 'orange']

# Print value of list at index 3

print(list\_fruits[3]);

Output:

Traceback (most recent call last):

  File "list-index.py", line 2, in <module>

    print(list\_fruits[3]);

IndexError: list index out of range

In the above example, we have created a list named “list\_fruits” with three values apple, banana, and orange. Here we are trying to print the value at the index [3].

And we know that the index of a list starts from 0 that’s why in the list, the last index is 2, not 3.

Due to which if we try to print the value at index [3] it will give an error.

Correct Example:

# Declaring list

list\_fruits = ['Apple', 'Banana', 'Orange']

# Print list element at index 2

print(list\_fruits[2]);

Output:

Orange

Q6. If you pass a list to a function, and if you want the function to be able to change the values of the list—so that the list is different after the function returns—what action should you avoid?

Your lists get modified because lists are mutable objects that are

capable of being modified, and you modify them

Objects are not copied when you pass them to a function. Any  
modification you make to the object is visible outside the function.

In your example, you are using this:

x = [[2,2]]

y = [[3,3]]

then inside a function:

z = a + b

This gives you a new list z = [[2, 2,], [3, 3]] but the inner lists  
aren’t copies of the oiginal [2, 2] and [3, 3] lists, they are exactly  
the same lists. Any modifications you make to the inner lists of z are  
visible everywhere.

You wrote:

“So passing lists to a function where they’re used with different  
variable names causes them to be altered? Never expected that, but it  
certainly explains the weird results I’m getting. I thought Python  
functions were supposed to create new local-only variables.”

The variables (name bindings) are local only: assigning to a local  
name doesn’t change local names in other functions.

The contents of the variable are objects, not copies. If you modify  
the object, you have modified the object, not a copy of the object.

You don’t need anything as complicated as your example to demonstrate  
Python’s object behaviour:

L = []

def demo(a):

a.append(1)

demo(L)

print(L)

What do you expect this to do? If you expect it to print [1], then you  
expect exactly the behaviour which Python uses. If you expect it to  
print the empty list [] then you will be disappointed.

You ask:

“how I can prevent my lists from being altered when passed to a  
function?”

Either re-think your strategy, or make a copy of the lists (either  
before passing them in, or after passing them in but before modifying  
them).

For this specific example, the simplest change you could make is to  
make a deep copy of z (a shallow copy isn’t sufficient):

import copy

# inside your function

z = copy.deepcopy(a + b)

but that gets very expensive for arbitrarily large and deeply nested  
objects. An alternative would be:

z = [x[:] for x in a] # Copy the sublists in a.

z.extend(x[:] for x in b) # Copy the sublists in b.

but if you have three or more levels of nesting it won’t copy those.

Q7. What is the concept of an unbalanced matrix?

Whenever the cost matrix of an assignment problem is not a square matrix, that is, whenever the number of sources is not equal to the number of destinations, the assignment problem is called an unbalanced assignment problem. In such problems, dummy rows (or columns) are added in the matrix so as to complete it to form a square matrix. The dummy rows or columns will contain all costs elements as zeroes. The Hungarian method may be used to solve the problem.

Example: Unbalanced Assignment Problem

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Job | | | | |
| Person | 1 | 2 | 3 | 4 |
| A | 20 | 25 | 22 | 28 |
| B | 15 | 18 | 23 | 17 |
| C | 19 | 17 | 21 | 24 |

Solution

Since the number of persons is less than the number of jobs, we introduce a dummy person (D) with zero values. The revised assignment problem is given below:

Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Job | | | | |
| Person | 1 | 2 | 3 | 4 |
| A | 20 | 25 | 22 | 28 |
| B | 15 | 18 | 23 | 17 |
| C | 19 | 17 | 21 | 24 |
| D (dummy) | 0 | 0 | 0 | 0 |

Q8. Why is it necessary to use either list comprehension or a loop to create arbitrarily large matrices?

Python features functional programming tools like map and filter for mapping operations over sequences and collecting results.

Since this is such a common task in Python coding, Python made a new expression: the list comprehension which is more flexible than map and filter. List comprehensions apply an arbitrary expression to items in an iterable rather than applying function. It provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

List Comprehension vs. map

Python's built-in ord returns the ASCII integer code of a character:

>>> ord('A')

65

If we want to collect the ASCII codes of all characters in a string, the most straightforward method is using a for loop and append the results to a list:

>>> result = []

>>> for x in 'Dostoyevsky':

result.append(ord(x))

>>> result

[68, 111, 115, 116, 111, 121, 101, 118, 115, 107, 121]

>>>

If we use map, we can get the same result with a single function call:

>>> result = list(map(ord,'Dostoyevsky'))

>>> result

[68, 111, 115, 116, 111, 121, 101, 118, 115, 107, 121]

>>>

But, we can get the similar result from a list comprehension expression. While map maps a function over a sequence, list comprehensions map an expression over a sequence:

>>> result = [ord(x) for x in 'Dostoyevsky']

>>> result

[68, 111, 115, 116, 111, 121, 101, 118, 115, 107, 121]

>>>

List comprehensions collect the result of applying an arbitrary expression to a sequence and return them in a new list. The effect is similar to that of the for loop and the map call. List comprehensions become more convenient when we need to apply an arbitrary expression to a sequence:

>>> [x \*\* 3 for x in range(5)]

[0, 1, 8, 27, 64]

If we had to use map, we would need to write a function to implement the square operation, probably, lambda instead of using a def:

>>> list(map((lambda x: x \*\* 2),range(5)))

[0, 1, 4, 9, 16]

This does the job. It's only a little bit longer that the list comprehension. For more advance kinds of expressions, however, list comprehensions will often require considerably less typing.

List Comprehension with filter

If we use if with list compression, it is almost equivalent to the filter built-in.

Let's make examples using both schemes.

>>>

>>> [x for x in range(10) if x % 2 == 0]

[0, 2, 4, 6, 8]

>>> list(filter((lambda x: x % 2 == 0), range(10)))

[0, 2, 4, 6, 8]

>>> result = []

>>> for x in range(10):

if x % 2 == 0:

result.append(x)

>>> result

[0, 2, 4, 6, 8]

>>>

All of these use the modulus operator %, to extract even numbers. The filter call here is not much longer that the list comprehension either. But we can combine an if and a map, in a single expression:

>>> [x \*\* 2 for x in range(10) if x % 2 == 0]

[0, 4, 16, 36, 64]

>>>

We collect the squares of the even numbers from 0 to 9. The for loop skips numbers which the if on the right is false. The expression on the left computes the squares. The equivalent map requires a lot more work. We have to combine filter with map iteration:

>>> list( map((lambda x: x \*\* 2), filter((lambda x: x % 2== 0),range(10))) )

[0, 4, 16, 36, 64]

Actually, list comprehensions are more general. We can code any number of nested for loop in a list comprehension, and each may have an optional associated if test. When for loop are nested within a list comprehension, they work like equivalent for loop statement:

>>> result = []

>>> result = [ x \*\* y for x in [10, 20, 30] for y in [2, 3, 4]]

>>> result

[100, 1000, 10000, 400, 8000, 160000, 900, 27000, 810000]

>>>