Python List Indexing and Slicing

Lists allow Python programs to store multiple values in one place.

But storing data is only the beginning.

To work with a list effectively, you need to know how to:

- access a specific item
- access items from the end
- extract a range of items
- skip items
- reverse a list using slicing
- update items through indexes
- replace parts of a list
- safely work with indexes

These operations are called **list indexing and slicing**.

---

1. A List Has Positions

Consider:

```
fruits = ["apple", "banana", "mango", "orange"]
```

Each item has a position called an **index**.

```
Index:   0       1       2       3
         □       □       □       □
```

```
    apple    banana    mango    orange
```

Python starts counting from **0**, not 1.

Therefore:

```
print(fruits[0])
```

Output:

```
apple
```

And:

```
print(fruits[3])
```

Output:

```
orange
```

---

2. The Indexing Syntax

The basic syntax is:

```
list[index]
```

Example:

```
languages = ["Python", "JavaScript", "Dart", "Java"]

print(languages[0])
```

```
print(languages[2])
```

Output:

```
Python
Dart
```

The index tells Python exactly which element you want.

---

3. Zero Based Indexing

Suppose you have:

```
numbers = [10, 20, 30, 40, 50]
```

The positions are:

```
Index:    0   1   2   3   4
          □   □   □   □   □
Value:   10  20  30  40  50
```

So:

```
numbers[0]  # 10
numbers[1]  # 20
numbers[2]  # 30
numbers[3]  # 40
numbers[4]  # 50
```

A common beginner mistake is thinking:

`numbers[1]`

means the first element.

It actually means the **second element**.

---

4. Why Does Python Start at 0?

Indexes represent an offset from the beginning.

The first item is zero positions away from the start.

```
First item  → offset 0
Second item → offset 1
Third item  → offset 2
```

This convention is used throughout programming and becomes especially important when working with loops, arrays, algorithms, and other data structures.

---

5. Negative Indexing

Python also lets you access items from the end of the list.

Example:

```
fruits = ["apple", "banana", "mango", "orange"]
```

Positive indexes:

```
Index:     0        1        2        3
           □        □        □        □
        apple    banana   mango    orange
```

Negative indexes:

```
Index:    -4       -3       -2       -1
           □        □        □        □
        apple    banana   mango    orange
```

Therefore:

```
print(fruits[-1])
```

Output:

```
orange
```

And:

```
print(fruits[-2])
```

Output:

```
mango
```

---

6. Why Negative Indexing Is Useful

Suppose a list contains many items:

```
resources = [
```

```
    "Python Basics",
    "Python Loops",
    "Python Strings",
    "Python Lists",
    "Python Functions"
]
```

If you want the last resource, you don't need to know its exact index.

Simply use:

```
print(resources[-1])
```

Output:

```
Python Functions
```

This is especially useful when the size of the list can change.

---

7. Accessing the First and Last Items

A very common pattern is:

```
first = items[0]
last = items[-1]
```

Example:

```
courses = ["Python", "JavaScript", "React", "Flutter"]

print("First:", courses[0])
```

```
print("Last:", courses[-1])
```

Output:

```
First: Python
Last: Flutter
```

---

8. IndexError

What happens if you access an index that doesn't exist?

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

print(fruits[5])
```

Python raises:

```
IndexError
```

Why?

The valid positive indexes are only:

```
0
1
2
```

The valid negative indexes are:

```
-1
-2
```

`-3`

There is no index `5`.

---

9. Finding the Valid Index Range

For a list of length `n`:

Positive indexes

`0 → n - 1`

Negative indexes

`-1 → -n`

For example, a list with 5 items has:

```
Positive: 0  1  2  3  4
Negative: -5 -4 -3 -2 -1
```

This is a useful rule to remember.

---

10. Using `len()` With Indexes

You can determine the length of a list with:

```
len(items)
```

Example:

```
numbers = [10, 20, 30, 40]

print(len(numbers))
```

Output:

```
4
```

The last positive index is:

```
len(numbers) - 1
```

So:

```
print(numbers[len(numbers) - 1])
```

Output:

```
40
```

However, when you simply want the last item, this is cleaner:

```
print(numbers[-1])
```

---

11. Updating a List Using an Index

Lists are mutable.

That means you can change an item using its index.

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

```
fruits[1] = "orange"

print(fruits)
```

Output:

```
['apple', 'orange', 'mango']
```

The item at index `1` was replaced.

---

12. Updating With Negative Indexes

Negative indexes can also modify items.

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

fruits[-1] = "orange"

print(fruits)
```

Output:

```
['apple', 'banana', 'orange']
```

Here, `-1` refers to the last item.

---

13. What Is Slicing?

Indexing gives you **one item**.

Slicing gives you **multiple items**.

Example:

```
numbers = [10, 20, 30, 40, 50]
```

Indexing:

```
print(numbers[2])
```

Output:

```
30
```

Slicing:

```
print(numbers[1:4])
```

Output:

```
[20, 30, 40]
```

So:

> **Indexing selects an item. Slicing selects a portion of a list.**

---

14. Basic Slicing Syntax

The basic syntax is:

```
list[start:end]
```

For example:

```
numbers = [10, 20, 30, 40, 50]

print(numbers[1:4])
```

Output:

```
[20, 30, 40]
```

The most important rule is:

```
start □ included
end   □ excluded
```

So:

```
numbers[1:4]
```

includes indexes:

```
1
2
3
```

but not:

```
4
```

---

15. Visualizing Slicing

Consider:

```
numbers = [10, 20, 30, 40, 50]

Index:    0    1    2    3    4
          □    □    □    □    □
Value:   10   20   30   40   50
          □□□□□□□□□□□□□□□□
           [1:4]
```

The result is:

```
[20, 30, 40]
```

Think of the ending index as a **stop position**, not an item to include.

---

16. Slicing From the Beginning

You can leave out the start index.

```
numbers = [10, 20, 30, 40, 50]

print(numbers[:3])
```

Output:

```
[10, 20, 30]
```

This means:

```
numbers[0:3]
```

So:

```
[:3]  start from the beginning
```

---

17. Slicing to the End

You can leave out the end index.

```
numbers = [10, 20, 30, 40, 50]

print(numbers[2:])
```

Output:

```
[30, 40, 50]
```

This means:

```
start at index 2
continue until the end
```

---

18. Copying a List With Slicing

Using:

`items[:]`

creates a shallow copy.

Example:

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

copy = numbers[:]

copy.append(4)

print(numbers)
print(copy)
```

Output:

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

This is different from:

```
copy = numbers
```

because assignment creates another reference to the same list.

---

19. Slicing With Negative Indexes

You can combine slicing with negative indexes.

```
numbers = [10, 20, 30, 40, 50]
```

```
print(numbers[-3:])
```

Output:

```
[30, 40, 50]
```

This means:

> Start from the third item from the end and continue to the end.

Another example:

```
print(numbers[:-2])
```

Output:

```
[10, 20, 30]
```

This means:

> Take everything except the last two items.

---

20. Useful Negative Slices

For:

```
numbers = [10, 20, 30, 40, 50]
```

Last item

```
numbers[-1:]
```

Result:

```
[50]
```

Last two items

```
numbers[-2:]
```

Result:

```
[40, 50]
```

Everything except last item

```
numbers[:-1]
```

Result:

```
[10, 20, 30, 40]
```

Everything except last two

```
numbers[:-2]
```

Result:

```
[10, 20, 30]
```

These patterns appear frequently in real Python code.

---

21. The Step in Slicing

Slicing has a third component:

`list[start:end:step]`

For example:

```
numbers = [0, 1, 2, 3, 4, 5, 6]

print(numbers[0:7:2])
```

Output:

```
[0, 2, 4, 6]
```

The `2` means:

> Move two positions at a time.

---

22. Every Second Item

You don't need to specify start and end.

```
numbers = [0, 1, 2, 3, 4, 5, 6]

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

Output:

```
[0, 2, 4, 6]
```

This means:

```
start □ beginning
end   □ end
step  □ 2
```

---

23. Every Third Item

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8]

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

Output:

```
[0, 3, 6]
```

---

24. Negative Step

A negative step moves through the list backward.

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

print(numbers[::-1])
```

Output:

```
[5, 4, 3, 2, 1]
```

This is one of the most useful slicing patterns in Python.

---

25. Reversing a List With Slicing

Suppose:

```
resources = [
    "Python",
    "JavaScript",
    "React",
    "Flutter"
]
```

You can reverse the order:

```
reversed_resources = resources[::-1]

print(reversed_resources)
```

Output:

```
['Flutter', 'React', 'JavaScript', 'Python']
```

Notice an important difference:

```
resources[::-1]
```

creates a reversed list without changing the original list.

Whereas:

`resources.reverse()`

modifies the original list.

---

26. `reverse()` vs `[::-1]`

Operation	Original list changed?	Returns reversed list?
`reverse()`	Yes	No useful list return
`[::-1]`	No	Yes

Example:

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

reversed_numbers = numbers[::-1]

print(numbers)
print(reversed_numbers)
```

Output:

```
[1, 2, 3]
```

```
[3, 2, 1]
```

---

27. Slicing With a Negative Step

You can control the start and end while moving backward.

```
numbers = [0, 1, 2, 3, 4, 5]

print(numbers[5:1:-1])
```

Output:

```
[5, 4, 3, 2]
```

Remember:

The end index is still excluded.

So index `1` is not included.

---

28. Important Rule for Negative Steps

When the step is negative, Python moves from right to left.

For example:

```
numbers[4:1:-1]
```

means:

Start at index 4
↓
Move backward
↓
Stop before index 1

Result:

[4, 3, 2]

This becomes much easier when you visualize the indexes rather than trying to memorize patterns.

---

29. Empty Slices

Some slices produce an empty list.

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

print(numbers[3:2])
```

Output:

```
[]
```

Why?

The default step is positive, so Python tries to move forward from index `3` toward index `2`.

That direction doesn't work.

With a negative step:

```
print(numbers[3:2:-1])
```

Output:

```
[4]
```

The direction matters.

---

30. Slicing Does Not Usually Raise IndexError

This is an important difference between indexing and slicing.

Indexing:

```
numbers = [10, 20, 30]

print(numbers[10])
```

causes:

```
IndexError
```

But slicing:

```
print(numbers[1:10])
```

does not raise an error.

Output:

```
[20, 30]
```

Python simply takes the available elements.

This makes slicing useful when the requested range may extend beyond the list's actual size.

---

31. Replacing a Slice

Because lists are mutable, you can replace a slice.

```
numbers = [10, 20, 30, 40, 50]

numbers[1:3] = [200, 300]

print(numbers)
```

Output:

```
[10, 200, 300, 40, 50]
```

You can replace the slice with a different number of items.

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

numbers[1:3] = [20, 30, 40]

print(numbers)
```

Output:

```
[1, 20, 30, 40, 4]
```

The list grew because three values replaced two.

---

32. Removing a Slice

You can delete part of a list by assigning an empty list.

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

numbers[1:4] = []

print(numbers)
```

Output:

```
[1, 5]
```

You can also use `del`:

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

del numbers[1:4]

print(numbers)
```

Output:

```
[1, 5]
```

---

33. Inserting Through Slicing

A slice can also be replaced with more items.

For example:

```
numbers = [1, 4, 5]

numbers[1:1] = [2, 3]

print(numbers)
```

Output:

```
[1, 2, 3, 4, 5]
```

The empty slice at index `1` acts as an insertion point.

---

34. Step Slices and Assignment

When using an extended slice such as:

```
numbers[::2]
```

the replacement must have the correct number of elements.

Example:

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

numbers[::2] = [10, 30, 50]

print(numbers)
```

Output:

```
[10, 2, 30, 4, 50]
```

There are three selected positions:

```
index 0
index 2
index 4
```

so three replacement values are required.

---

35. Indexing Nested Lists

Lists can contain other lists.

```
students = [
    ["Hafsa", 90],
    ["Ali", 85],
    ["Sara", 95]
]
```

To access the first inner list:

```
print(students[0])
```

Output:

```
['Hafsa', 90]
```

To access the student's name:

```
print(students[0][0])
```

Output:

```
Hafsa
```

To access the score:

```
print(students[0][1])
```

Output:

```
90
```

Think of this as:

```
students
  
[0]  first student
  
[0]  name
```

---

36. Slicing Nested Lists

You can slice the outer list:

```
students = [
    ["Hafsa", 90],
    ["Ali", 85],
    ["Sara", 95],
    ["Ahmed", 88]
]
```

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

Output:

```
[['Hafsa', 90], ['Ali', 85]]
```

You can also slice an inner list:

```
student = ["Hafsa", 90, "Python"]

print(student[:2])
```

Output:

```
['Hafsa', 90]
```

---

37. Indexing While Looping

Sometimes you need both the item and its index.

You can use:

```
numbers = [10, 20, 30]

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

Output:

```
0 10
1 20
```

```
2 30
```

A cleaner approach is usually:

```
for index, number in enumerate(numbers):
    print(index, number)
```

The important concept is that indexes allow you to access and modify specific positions.

---

38. Real Example: Recent haas.dev Resources

Imagine your resources are stored in order from oldest to newest:

```
resources = [
    "Python Basics",
    "Python Loops",
    "Python Strings",
    "Python Lists",
    "Python Functions",
    "Python Dictionaries"
]
```

To display the three most recent resources:

```
recent = resources[-3:]

print(recent)
```

Output:

```
['Python Lists', 'Python Functions', 'Python Dictionaries']
```

No matter how many resources are added later, the same slice can still retrieve the last three.

---

39. Real Example: Pagination

Suppose you have many resources:

```
resources = [
    "Resource 1",
    "Resource 2",
    "Resource 3",
    "Resource 4",
    "Resource 5",
    "Resource 6",
    "Resource 7",
    "Resource 8"
]
```

You want to display four at a time.

First page:

```
page_1 = resources[0:4]
```

Result:

```
['Resource 1', 'Resource 2', 'Resource 3', 'Resource 4']
```

Second page:

```
page_2 = resources[4:8]
```

Result:

```
['Resource 5', 'Resource 6', 'Resource 7', 'Resource 8']
```

This is the basic idea behind slicing data into pages.

---

40. Real Example: Processing Batches

Suppose you have:

```
users = [
    "Ali",
    "Sara",
    "Hafsa",
    "Ahmed",
    "Zara",
    "Hamza"
]
```

You can process three users at a time:

```
batch = users[:3]

print(batch)
```

Output:

```
['Ali', 'Sara', 'Hafsa']
```

Next batch:

```
batch = users[3:6]

print(batch)
```

Output:

```
['Ahmed', 'Zara', 'Hamza']
```

The same idea appears in data processing and API workflows.

---

41. Real Example: Extracting a Username

Suppose:

```
username = "hafsa_waseem"
```

You want the first five characters:

```
print(username[:5])
```

Output:

```
hafsa
```

You can also take the last five:

```
print(username[-5:])
```

Output:

```
aseem
```

---

42. Real Example: Recent Activity

Imagine:

```
activities = [
    "Login",
    "Viewed Python",
    "Downloaded PDF",
    "Completed Quiz",
    "Opened Resource"
]
```

To show only the last three:

```
recent_activity = activities[-3:]

for activity in recent_activity:
    print(activity)
```

Output:

```
Downloaded PDF
Completed Quiz
Opened Resource
```

This is a practical use of negative slicing.

---

43. Common Mistakes

Mistake 1: Forgetting that the end is excluded

```
numbers = [10, 20, 30, 40, 50]

print(numbers[1:4])
```

The result is:

```
[20, 30, 40]
```

not:

```
[20, 30, 40, 50]
```

Remember:

> Start included, end excluded.

Mistake 2: Confusing indexing and slicing

Indexing:

```
numbers[2]
```

returns one value:

```
30
```

Slicing:

```
numbers[2:3]
```

returns a list:

```
[30]
```

The brackets look similar, but the result type is different.

Mistake 3: Forgetting zero-based indexing

```
numbers = [10, 20, 30]
```

This:

```
numbers[1]
```

returns:

```
20
```

not `10`.

Mistake 4: Using the wrong direction

This:

```
numbers[1:4:-1]
```

does not produce the expected backward sequence because the step is negative while the start is before the end.

When using a negative step, make sure your start and end positions match the direction of travel.

Mistake 5: Assuming slicing modifies the original list

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

part = numbers[1:3]

print(numbers)
```

The original list remains:

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

Slicing creates a new list.

---

44. Indexing vs Slicing

Feature	Indexing	Slicing
Syntax	`items[2]`	`items[1:4]`
Returns	One item	A new list
Selects	One position	Range of positions
End included?	Not applicable	No

Supports negative indexes	Yes	Yes
Supports step	No	Yes
Can modify list	Yes, with assignment	Yes, with slice assignment

Example:

`items[2]`

means:

> Give me the item at position 2.

While:

`items[1:4]`

means:

> Give me items from position 1 up to, but not including, position 4.

---

45. List Slicing Cheat Sheet

Given:

`items = [0, 1, 2, 3, 4, 5]`

First item

`items[0]`

Last item

`items[-1]`

First three items

`items[:3]`

From index 2 to the end

`items[2:]`

Last three items

`items[-3:]`

Everything except the last item

`items[:-1]`

Every second item

`items[::2]`

Every second item starting from index 1

`items[1::2]`

Reverse the list

`items[::-1]`

Reverse a portion

`items[4:1:-1]`

Copy the list

`items[:]`

---

46. A Better Way to Read Slices

Whenever you see:

`items[start:end:step]`

ask three questions:

Where do I start?

`start`

Where do I stop?

`end`

Remember that the end is excluded.

Which direction and how far do I move?

`step`

For example:

`items[1:6:2]`

means:

`Start → 1`
`Stop → before 6`
`Step → 2`

Selected indexes:

`1 → 3 → 5`

This mental model is much easier than memorizing individual slicing patterns.

---

47. Mini Project: Recent Resource Viewer

Build a small program that stores resource titles and lets the user choose how many recent resources to view.

```
resources = [
    "Python Basics",
    "Python Conditions",
    "Python Loops",
    "Python Strings",
    "Python Lists",
    "Python Functions",
    "Python Dictionaries"
]

count = int(input("How many recent resources do you want to see? "))

if count <= 0:
    print("Enter a positive number.")
```

```
else:
    recent = resources[-count:]

    print("\nRecent Resources:")

    for number, resource in enumerate(recent, start=1):
        print(f"{number}. {resource}")
```

If the user enters:

```
3
```

Output:

```
Recent Resources:
1. Python Lists
2. Python Functions
3. Python Dictionaries
```

This small project uses:

- lists
- negative indexing
- slicing
- len() concepts
- loops
- enumerate()
- user input

---

48. Practice Exercises

Exercise 1: First and Last

Given:

```
languages = ["Python", "JavaScript", "Java", "Dart", "Swift"]
```

Print:

- first language
- last language

Exercise 2: Middle Items

Given:

```
numbers = [10, 20, 30, 40, 50, 60]
```

Use slicing to produce:

```
[20, 30, 40, 50]
```

Exercise 3: Last Three

Given:

```
courses = [
    "Python",
    "JavaScript",
    "React",
    "Flutter",
```

```
    "Django"
]
```

Use negative slicing to get the last three courses.

Exercise 4: Every Second Item

Given:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

Produce:

```
[1, 3, 5, 7]
```

using slicing.

Exercise 5: Reverse

Reverse:

```
items = ["A", "B", "C", "D", "E"]
```

using slicing only.

Exercise 6: Remove Last Two

Given:

```
numbers = [10, 20, 30, 40, 50, 60]
```

Use slicing to produce:

```
[10, 20, 30, 40]
```

Exercise 7: Replace a Slice

Given:

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

Replace:

```
2, 3
```

with:

```
20, 30
```

Exercise 8: Nested Lists

Given:

```
students = [
    ["Hafsa", 90],
    ["Ali", 85],
    ["Sara", 95]
]
```

Use indexing to print:

```
Hafsa
95
```

---

49. 5 Minute Challenge

Create a **Recent Downloads Viewer**.

Start with:

```
downloads = [
    "Python Basics.pdf",
    "Python Loops.pdf",
    "Python Strings.pdf",
    "Python Lists.pdf",
    "Python Functions.pdf",
    "Python Dictionaries.pdf"
]
```

Ask the user:

```
How many recent downloads should be displayed?
```

Then:

1. Use negative slicing.
2. Display only that number of recent downloads.
3. Number the results using `enumerate()`.
4. Handle a number larger than the list length gracefully.
5. Display the original list afterward to prove it wasn't modified.

---

50. Mini Quiz

Question 1

What does this return?

```
numbers = [10, 20, 30, 40]
numbers[1]
```

A. 10
B. 20
C. 30
D. [20]

Answer: B

Question 2

What does this return?

```
numbers = [10, 20, 30, 40, 50]

numbers[1:4]
```

A. [10, 20, 30]
B. [20, 30, 40]
C. [20, 30, 40, 50]
D. [10, 20, 30, 40]

Answer: B

Question 3

What does this do?

```
numbers[::-1]
```

A. Sorts the list
B. Removes the list
C. Reverses the list into a new list
D. Returns the first item

Answer: C

---

51. Interview Questions

Beginner

1. What is list indexing?
2. Why does Python use zero-based indexing?
3. How do you access the first item of a list?
4. How do you access the last item?
5. What is negative indexing?
6. What happens when you access an invalid index?
7. What is list slicing?
8. What is the difference between `items[2]` and `items[2:3]`?
9. Is the end index included in a slice?

10. How do you get the first three elements?

Intermediate

11. How do you get the last three elements using slicing?
12. How do you reverse a list using slicing?
13. What does the third value in `start:end:step` represent?
14. What happens when the step is negative?
15. Why doesn't slicing usually raise `IndexError` when the end is beyond the list?
16. What is the difference between `reverse()` and `[::-1]`?
17. How can you create a shallow copy using slicing?
18. How can slice assignment be used to replace multiple elements?
19. How can slicing be used to remove elements?
20. How would you use slicing to implement simple pagination?

---

52. Key Takeaways

- List indexes start at `0`.
- Negative indexes count from the end.
- `items[0]` accesses the first item.
- `items[-1]` accesses the last item.
- Invalid indexes can raise `IndexError`.
- Slicing extracts multiple items.
- The basic syntax is `items[start:end]`.
- The start is included.
- The end is excluded.
- The full syntax is `items[start:end:step]`.
- A positive step moves forward.
- A negative step moves backward.
- `items[::-1]` creates a reversed copy.

- `items[:]` creates a shallow copy.
- Slicing usually creates a new list rather than changing the original.
- Slice assignment can modify, replace, insert, or delete sections of a list.
- Indexing and slicing also work with nested lists.
- Negative slicing is especially useful for recent or last-N items.
- Understanding indexes and slices is essential for loops, data processing, algorithms, and real application development.

Final Mental Model

Think of a list as a row of boxes:

```
  0    1    2    3    4
  □    □    □    □    □
┌────┬────┬────┬────┬────┐
│ 10 │ 20 │ 30 │ 40 │ 50 │
└────┴────┴────┴────┴────┘
 -5   -4   -3   -2   -1
```

Indexing

`items[2]`

means:

> Give me one box.

Slicing

`items[1:4]`

means:

> Give me a range of boxes from index 1 up to, but not including, index 4.

Step

`items[::2]`

means:

> Move through the boxes two positions at a time.

Reverse

`items[::-1]`

means:

> Move through the boxes from right to left.

Once this mental model becomes natural, list indexing and slicing become one of the most powerful and flexible parts of Python.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app/resources