

CSCI 145 · Introduction to Computing · Fall 2026

Lists

Lecture 8 — creating, changing, and looping over lists

Outline

- What is a list?

- Changing a list

- Lists and loops

- Named constants

Part 1 of 4

What is a list?

A list holds many values

A **list** is an ordered sequence of values, written inside square brackets

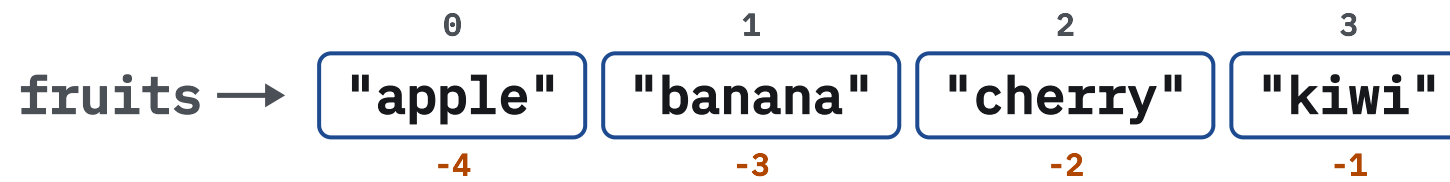
```
fruits = ["apple", "banana", "cherry"]
```

fruits → "apple" "banana" "cherry"

- Items are separated by commas
- Items can be any type, even mixed: `[3, "hi", 2.5, True]`
- The **empty list** is `[]`

Indexing: same rules as strings

```
fruits = ["apple", "banana", "cherry", "kiwi"]
```



- The first item is at index 0
- Negative indices count from the end
-1 is the last item
- The last index is `len(fruits) - 1`

```
>>> fruits[0]
'apple'
>>> fruits[2]
'cherry'
>>> fruits[-1]
'kiwi'
>>> len(fruits)
4
>>> fruits[4]
IndexError: list index
out of range
```

Slicing: a slice of a list is a list

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



- Like `s[1:3]` on a string
Start is included, stop is not
- Each slice is a **new** list
- The original list doesn't change

```
>>> nums[1:3]
[20, 30]
>>> nums[:2]
[10, 20]
>>> nums[3:]
[40, 50]
>>> nums
[10, 20, 30, 40, 50]
```

Lists work a lot like strings

```
s = "cat" and nums = [4, 2, 7]
```

	String	List
Length	<code>len(s) → 3</code>	<code>len(nums) → 3</code>
Index	<code>s[0] → "c"</code>	<code>nums[0] → 4</code>
Slice	<code>s[1:] → "at"</code>	<code>nums[1:] → [2, 7]</code>
Join with <code>+</code>	<code>s + "s" → "cats"</code>	<code>nums + [9] → [4, 2, 7, 9]</code>
Repeat with <code>*</code>	<code>s * 2 → "catcat"</code>	<code>[0] * 3 → [0, 0, 0]</code>
Check with <code>in</code>	<code>"a" in s → True</code>	<code>7 in nums → True</code>
Loop with <code>for</code>	<code>for ch in s:</code>	<code>for x in nums:</code>

One big difference: you can **change** a list

Part 2 of 4

Changing a list

Lists are mutable, strings are not

A string is immutable

```
>>> s = "cat"
>>> s[0] = "b"
TypeError: 'str' object does not
support item assignment
```

A list is mutable

```
>>> pets = ["cat", "dog"]
>>> pets[0] = "bird"
>>> pets
['bird', 'dog']
```

pets → "bird" "dog"

- Assign to an index to **replace** that item
- The list changes **in place**: same list, new contents

Adding items: `append` and `insert`

CODE

```
nums = [4, 2, 3]
```

```
nums.append(5)
```

```
nums.insert(1, 9)
```

NUMS AFTER

0	1	2
4	2	3

0	1	2	3
4	2	3	5

0	1	2	3	4
4	9	2	3	5

- `nums.append(x)` adds `x` to the **end**
- `nums.insert(i, x)` puts `x` at index `i`
Later items shift right
- These are **methods**: functions called with a dot after the list
- `append` and `insert` don't return anything
They change the list they are called on

String methods vs. list methods

`str.upper` returns a new string

```
>>> s = "hi"  
>>> s.upper()  
'HI'  
>>> s  
'hi'
```

- Strings are immutable
So string methods build a new string
- `append` and `insert` return `None`
The list itself changes

`list.append` changes the list

```
>>> nums = [4, 2]  
>>> nums.append(3)  
>>> nums  
[4, 2, 3]
```

```
>>> x = nums.append(6)  
>>> nums  
[4, 2, 3, 6]  
>>> print(x)  
None
```

Removing items: `remove` and `pop`

CODE

```
nums = [4, 2, 3, 2]
```

```
nums.remove(2)
```

```
nums.pop()
```

```
nums.pop(0)
```

RETURNS

—

—

2

4

NUMS AFTER

0	1	2	3
4	2	3	2

0	1	2
4	3	2

0	1
4	3

0
3

- `nums.remove(x)` removes the **first** `x`
- `nums.pop()` removes the last item and **returns** it
- `nums.pop(i)` removes and returns the item at index `i`

List methods documentation: docs.python.org/3/tutorial/datastructures.html#more-on-lists

Exercise: what is printed?

```
nums = []  
nums.append(7)  
nums.append(3)  
nums.insert(0, 5)  
nums[-1] = 10  
print(nums, len(nums))
```

NUMS AFTER

[]

7

7 3

5 7 3

5 7 10

OUTPUT

[5, 7, 10] 3

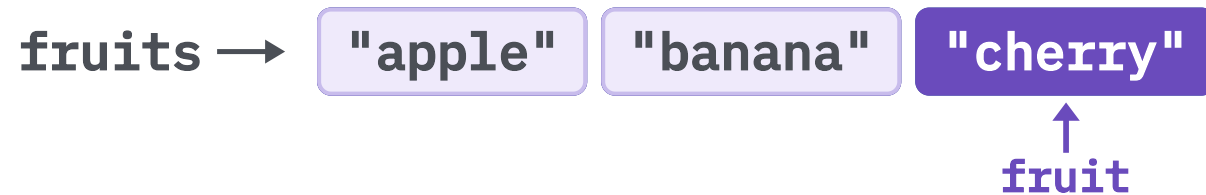
Part 3 of 4

Lists and loops

A **for** loop visits each item

Just like a string: the loop variable gets each item in order

```
fruits = ["apple", "banana", "cherry"]  
for fruit in fruits:  
    print(fruit)
```



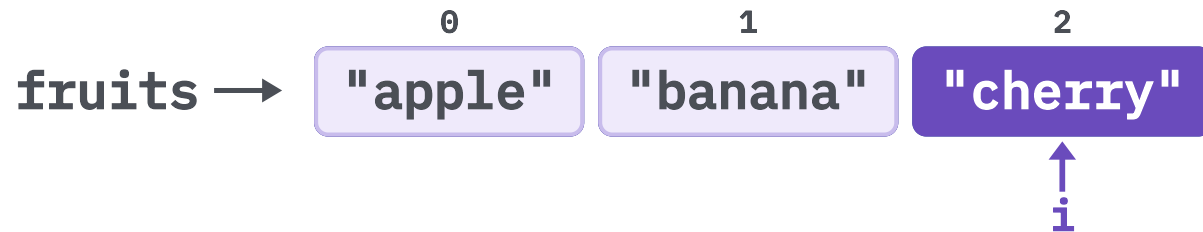
OUTPUT

```
apple  
banana  
cherry
```

When you need the index too

Number the items: loop over the **indices** with `range(len(fruits))`

```
fruits = ["apple", "banana", "cherry"]  
for i in range(len(fruits)):  
    print(i + 1, fruits[i])
```



OUTPUT

```
1 apple  
2 banana  
3 cherry
```

- `i` is the position, `fruits[i]` is the item
- You also need the index to **change** items: `fruits[i] = ...`

Building a list: start with []

The accumulator pattern again: `get_factors(6)` returns `[1, 2, 3, 6]`

```
def get_factors(n):  
    factors = []  
    for i in range(1, n + 1):  
        if n % i == 0:  
            factors.append(i)  
    return factors
```

- Before the loop: start with an **empty list**
- Inside the loop: `.append` each item

i	n % i == 0	factors
-	-	[]
1	True	[1]
2	True	[1, 2]
3	True	[1, 2, 3]
4	False	[1, 2, 3]
5	False	[1, 2, 3]
6	True	[1, 2, 3, 6]

What does `keep_evens([3, 4, 5])` return?

```
def keep_evens(nums):  
    evens = []  
    for num in nums:  
        if num % 2 == 0:  
            evens.append(num)  
    return evens
```

Go to **slido.com** and enter the code: `3674841`

Or [join directly here](#)



What does `keep_evens([3, 4, 5])` return?

```
def keep_evens(nums):  
    evens = []  
    for num in nums:  
        if num % 2 == 0:  
            evens = evens.append(num)  
    return evens
```

Returns None, not [4]

`append` changes the list and returns `None`
So `evens` becomes `None`

```
def keep_evens(nums):  
    evens = []  
    for num in nums:  
        if num % 2 == 0:  
            evens.append(num)  
    return evens
```

✓ Fix: just call `append`

Don't assign its result
Now it returns `[4]`

Built-in functions for lists of numbers

```
>>> scores = [88, 92, 75, 100]
>>> sum(scores)
355
>>> min(scores)
75
>>> max(scores)
100
>>> sum(scores) / len(scores)
88.75
```

- `sum` adds up the items
- `min` finds the smallest
`max` finds the largest
- Average: `sum` divided by `len`
- Lab 3: `min` and `max` of a list of times

How could we write `max` ourselves?

Keep the **best so far**, and replace it when you find something better

Trace `highest(scores)` with `scores = [88, 92, 75, 100]`

```
def highest(scores):  
    best = scores[0]  
    for s in scores:  
        if s > best:  
            best = s  
    return best
```

s	s > best	best
–	–	88
88	False	88
92	True	92
75	False	92
100	True	100

Why start `best` at `scores[0]` ?

Starting at 0 would return 0
if every score were negative

Worksheet: trace loops on paper

1. Take a worksheet and work with a partner.
2. For each problem, fill in one row per pass through the loop.
3. Then write what the code returns or prints.
4. Don't run the code: the point is to trace it by hand.

Worksheet: [day08-worksheet.pdf](#) · Answer key: [day08-worksheet-key.pdf](#)

Worksheet answers: `count_e("eel")`

VERSION A

```
def count_e(word):  
    total = 0  
    for ch in word:  
        if ch == "e":  
            total += 1  
    return total
```

ch	total
-	0
"e"	1
"e"	2
"l"	2

Returns 2

VERSION B

```
def count_e(word):  
    for ch in word:  
        total = 0  
        if ch == "e":  
            total += 1  
    return total
```

ch	total
-	-
"e"	1
"e"	1
"l"	0

Returns 0

Version B resets `total` on every pass

The same mistake as the `count_a` poll in Lecture 7

Worksheet answers: problems 2 and 3

PROBLEM 2

```
def doubles(nums):  
    result = []  
    for num in nums:  
        if num > 2:  
            result.append(num * 2)  
    return result  
  
doubles([3, 1, 4])
```

num	num > 2	result
–	–	[]
3	True	[6]
1	False	[6]
4	True	[6, 8]

Returns [6, 8]

PROBLEM 3

```
nums = [4, 7, 1]  
for i in range(len(nums)):  
    nums[i] = nums[i] + i  
print(nums)
```

i	nums[i] + i	nums
–	–	[4, 7, 1]
0	4 + 0 = 4	[4, 7, 1]
1	7 + 1 = 8	[4, 8, 1]
2	1 + 2 = 3	[4, 8, 3]

Prints [4, 8, 3]

Part 4 of 4

Named constants

Magic numbers

LECTURE 7'S VERSION

```
def random_problem():  
    a = random.randint(1, 10)  
    op = random.choice("+ - *")  
    b = random.randint(1, 10)  
    return str(a) + " " + op + " " + str(b)
```

WITH NAMED CONSTANTS

```
MIN_NUM = 1  
MAX_NUM = 10  
OPERATORS = "+ - *"  
  
def random_problem():  
    a = random.randint(MIN_NUM, MAX_NUM)  
    op = random.choice(OPERATORS)  
    b = random.randint(MIN_NUM, MAX_NUM)  
    return str(a) + " " + op + " " + str(b)
```

- A **magic number** is a value with no name
What does 10 mean? What if it changes?
- A **named constant** says what the value means
To change it, edit one line

Named constants

```
WEEKDAYS = ["Mon", "Tue", "Wed", "Thu", "Fri"]  
PASSING = 60
```

```
def is_weekday(day):  
    return day in WEEKDAYS
```

```
def passed(score):  
    return score >= PASSING
```

- ALL_CAPS names, by convention
- At the top of the file
Outside any function
- Never reassigned
Python trusts you on this
- Any type works, even a list

Homework 3 asks you to use constants instead of magic numbers

Activity

1. Download [day08_lists.py](#) and save it in your `cs145/class` folder.
2. Launch Thonny. From the File menu, open `cs145/class/day08_lists.py`.
3. For part 1, loop over a list in `count_passing` and `average`.
4. For part 2, build a new list with `.append` in `squares` and `positives`.
5. If you have time, complete `longest_word` without using `max`.

To do

- Review this slide deck and the corresponding notes: [Notes 8](#)
- [Homework 2](#) is due Thursday 10/8
- [Lab 3](#) is this Friday
- Quiz 4 (Lectures 7–8) is at the start of next class: [practice quiz](#)
- [Homework 3](#) is due Thursday 10/15

Next time

Lecture 9 — more lists