

Lists

- List syntax
- List operations
- copy.deepcopy
- range
- while-else
- for
- for-break-continue-else



`L[-index] == L[len(L)-index]`
Negative indexing from the right is Python specific, and is e.g. not supported in Java, C, C++

List operations

- List syntax `[value0, value1, ..., valuek-1]`
 - List indexing `L[index], L[-index]`
 - ★ ■ List **slices** `L[from : to], L[from : to : step]` or `L[slice(from, to, step)]`
 - ★ ■ Creating a copy of a list `L[:]` or `L.copy()`
 - ★ ■ List concatenation (creates new list) `X + Y`
 - ★ ■ List repetition (repeated concatenation with itself) `42 * L`
 - Length of list `len(L)`
 - Check if element is in list `e in L` (returns True or False)
 - Check if element is not in list `e not in L` (same as `not e in L`)
 - Index of first occurrence of element in list `L.index(e)`
 - Number of occurrences of element in list `L.count(e)`
 - `sum(L)` `min(L)` `max(L)`
- ★ = creates new list

sum (...)

Python 3.12
change

Python shell

```
> 1 - 1/3 - 1 + 1/3 # mathematically should be zero
| 5.551115123125783e-17 # but floats are imprecise
> L = [1, -1/3, -1, 1/3]
> L
| [1, -0.3333333333333333, -1, 0.3333333333333333] # mix of int and float
> sum(L)
| 5.551115123125783e-17
> sum([1.0, -1/3, -1.0, 1/3]) # all floats
| 5.551115123125783e-17 # Python 3.11
> sum([1.0, -1/3, -1.0, 1/3])
| 0.0 # Python 3.12 uses "Neumaier summation" to improve accuracy for floats
> sum([1, -1/3, -1, 1/3])
| 5.551115123125783e-17 # Python 3.12 loses accuracy when mixing int and float
> import math
> math.fsum([1, -1/3, -1, 1/3]) # math.fsum more accurate float sums
| 0.0
```



List modifiers (lists are mutable)

- Append an element to a list (`L` is modified) `L.append(42)`
- Extend list with elements (`X` is modified) `X.extend(Y)`
- Replace sublist by another list (length can differ) `X[i:j] = Y`
- Delete elements from list `del L[i]` or `del L[i:j:k]`
- Remove & return element at position `L.pop(i)`
- Remove first occurrence of element `L.remove(e)`
- Reverse list (`L` is modified) `L.reverse()`
- `L *= 42`
- `L.insert(i, x)` same as `L[i:i] = [x]`

Python shell

```
> x = [1, 2, 3, 4, 5]
> x[2:4] = [10, 11, 12]
> x
| [1, 2, 10, 11, 12, 5]
> x = [1, 2, 11, 5, 8]
> x[1:4:2] = ['a', 'b']
| [1, 'a', 11, 'b', 8]
```

Questions – What is x ?

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
x[2:8:3] = ['a', 'b']
```

- a) [1, 2, 'a', 'b', 5, 6, 7, 8, 9, 10]
- b) [1, 'a', 3, 4, 5, 6, 7, 'b', 9, 10]
- c) [1, 2, 3, 4, 5, 6, 7, 'a', 'b']
- d) [1, 2, 'a', 4, 5, 'b', 7, 8, 9, 10]
- e) ValueError
- f) Don't know

Questions – What is y ?

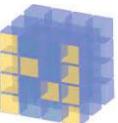
```
y = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]  
y = y[3:15:3][1:4:2]
```

- a) [3, 6, 9, 12, 15]
- b) [7, 13]
- c) [1, 9]
- d) [4, 7, 10, 13, 2, 4]
- e) TypeError
- f) Don't know

Nested lists (multi-dimensional lists)

- Lists can contain lists as elements, that can contain lists as elements, that ...
- Can e.g. be used to store multi-dimensional data (list lengths can be non-uniform)

Note: For dealing with matrices the NumPy module is a better choice



multidimensional-lists.py

```
list1d = [1, 3, 5, 2]
list2d = [[1, 2, 3, 4],
          [5, 6, 7, 9],
          [0, 8, 2, 3]]
list3d = [[[5,6], [4,2], [1,7], [2,4]],
          [[1,2], [6,3], [2,5], [7,5]],
          [[3,8], [1,5], [4,3], [2,4]]]
```

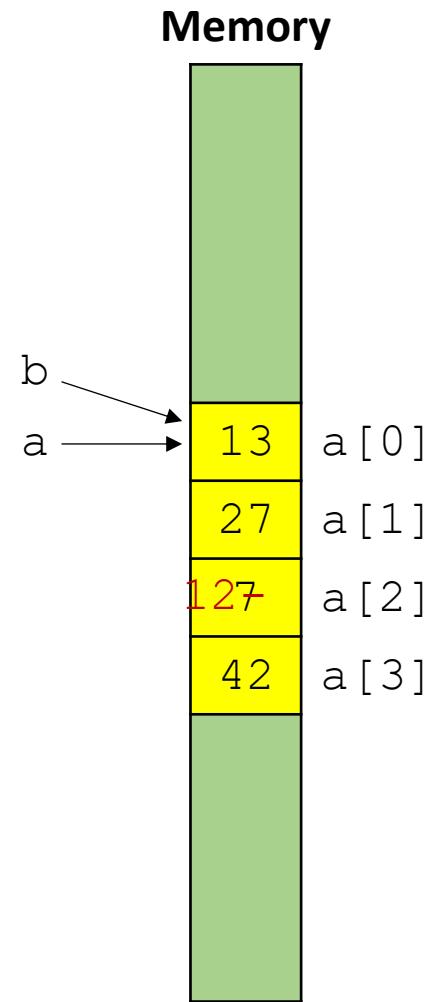
```
print(list1d[2])
print(list2d[1][2])
print(list3d[2][0][1])
```

Python shell

```
| 5
| 7
| 8
```

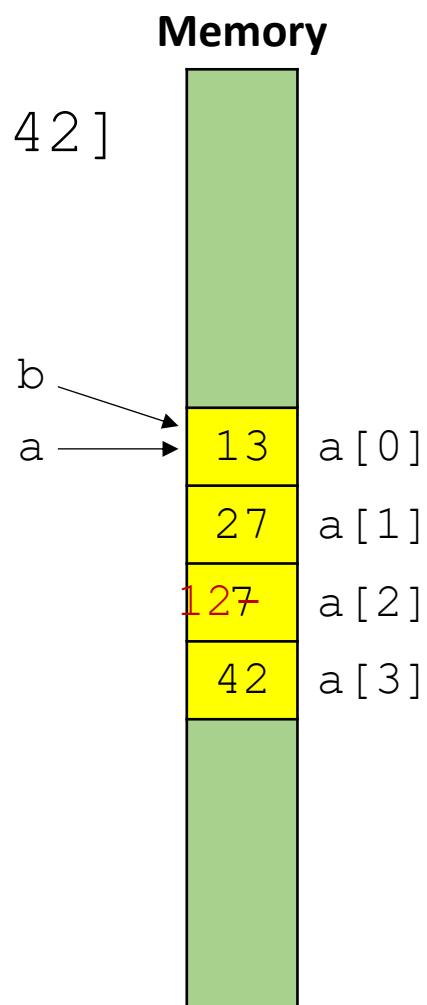
aliasing

```
a = [13, 27, 7, 42]  
b = a  
a[2] = 12
```

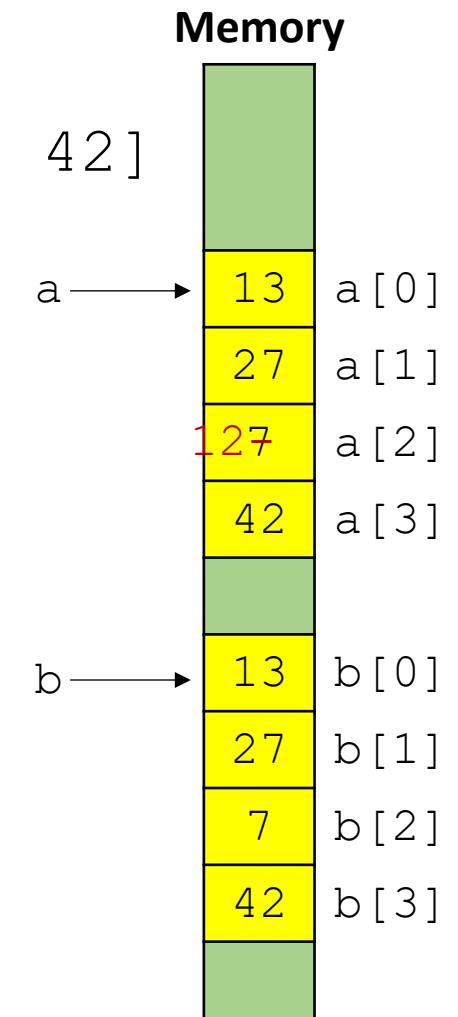


$y = x$ VS $y = x[:]$

$a = [13, 27, 7, 42]$
 $b = a$
 $a[2] = 12$



$a = [13, 27, 7, 42]$
 $b = a[:]$
 $a[2] = 12$



$\times [:]$ vs nested structures

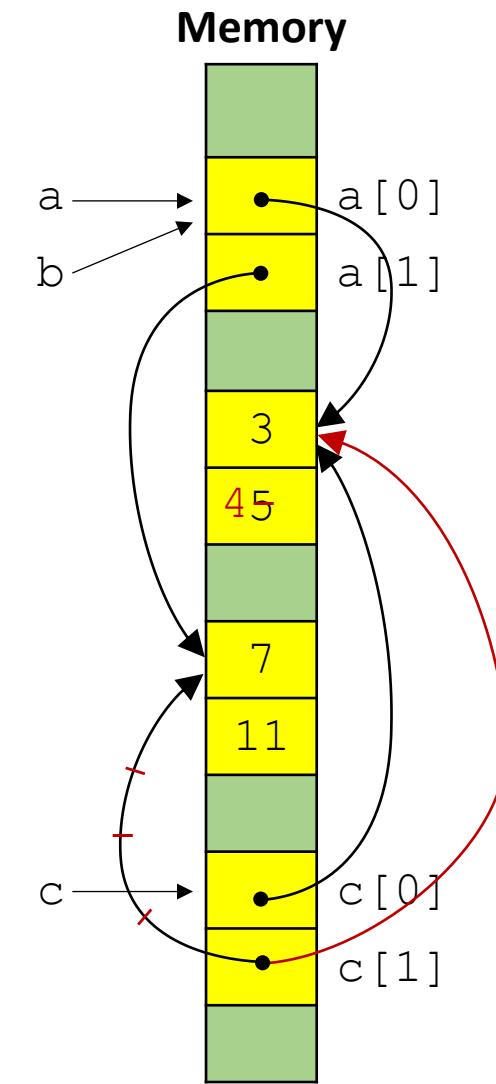
```
a = [[3, 5], [7, 11]]
```

```
b = a
```

```
c = a[:]
```

```
a[0][1] = 4
```

```
c[1] = b[0]
```



Question – what is c ?

- a) $\begin{bmatrix} [3, 5], [7, 11] \end{bmatrix}$
- b) $\begin{bmatrix} [3, 5], [3, 5] \end{bmatrix}$
- c) $\begin{bmatrix} [3, 4], [3, 5] \end{bmatrix}$
- d) $\begin{bmatrix} [3, 4], [3, 4] \end{bmatrix}$
- e) Don't know

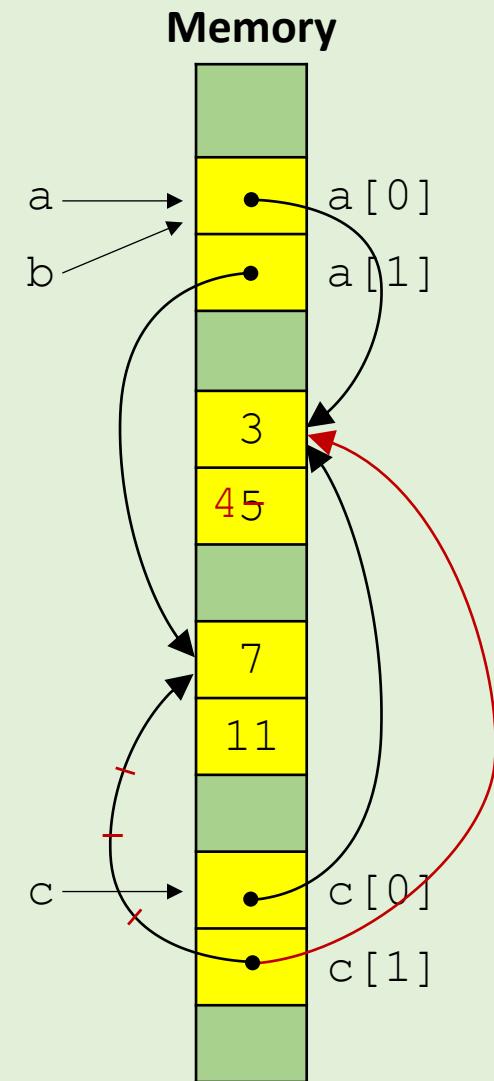
$a = \begin{bmatrix} [3, 5], [7, 11] \end{bmatrix}$

$b = a$

$c = a[:]$

$a[0][1] = 4$

$c[1] = b[0]$

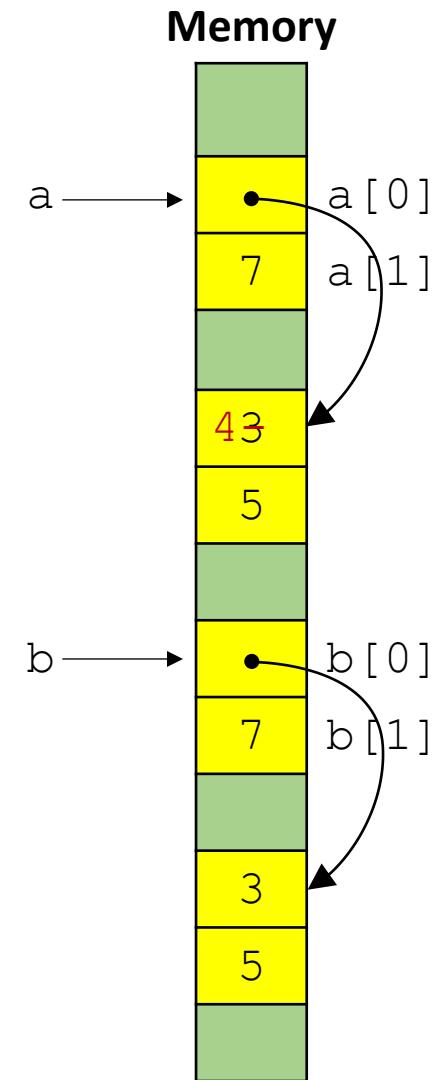


copy.deepcopy

- To make a copy of all parts of a composite value use the function **deepcopy** from module `copy`

Python shell

```
> from copy import deepcopy
> a = [[3, 5], 7]
> b = deepcopy(a)
> a[0][0] = 4
> a
| [[4,5],7]
> b
| [[3,5],7]
```



Initializing a 2-dimensional list



Python shell

```
> x = [1] * 3
> x
[1, 1, 1]
> y = [[1] * 3] * 4
> y
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
> y[0][0] = 0
> y
[[0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1]]
```

Python shell

```
> y = []
> for _ in range(4): y.append([1] * 3)
> y[0][0] = 0
> y
[[0, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

range(*from*, *to*, *step*)

 In Python 2, `range` generates the explicit list, i.e. always use memory proportional to the length; `xrange` in Python 2 corresponds to `range` in Python 3; Python 3 is more memory friendly

- `range` (*from*, *to*, *else*) represents the **sequence** of numbers starting with *from*, with increments of *step*, and smaller/greater than *to* if *step* positive/negative

<code>range(5)</code>	: 0, 1, 2, 3, 4	(default <i>from</i> = 0, <i>step</i> = 1)
<code>range(3, 8)</code>	: 3, 4, 5, 6, 7	(default <i>step</i> = 1)
<code>range(-2, 7, 3)</code>	: -2, 1, 4	(<i>from</i> and <i>to</i> can be any integer)
<code>range(2, -5, -2)</code>	: 2, 0, -2, -4	(decreasing sequence if <i>step</i> negative)

- Ranges are immutable, can be indexed like a list, sliced, and compared (i.e. represent the same sequence of numbers)
- `list(range(...))` generates the explicit list of numbers

Python shell

```
> range(1, 10000000, 3)[2]
| 7
> range(1, 10000000, 3)[100:120:4]
| range(301, 361, 12)
> range(1, 10000000, 3)[100:120:4][2:3]
| range(325, 337, 12)
> list(range(5, 14, 3))
| [5, 8, 11]
```

Question – What is `range(3, 20, 4)[2:4][1]` ?

- a) 3
- b) 7
- c) 11
- d) 15
- e) 19
- f) Don't know



Exercise “sliced ranges” is annoying hard

for - loop

- For every element in a **sequence** execute a block of code:

```
for variable in sequence:  
    block
```

- Sequences can e.g. be lists, strings, ranges
- `break` and `continue` can be used like in a while-loop to break out of the for-loop or continue with the next element in the sequence

Python shell

```
> for x in [1, 'abc', [2, 3], 5.0]:  
>     print(x)  
1  
abc  
[2, 3]  
5.0  
> for x in 'abc':  
>     print(x)  
a  
b  
c  
> for x in range(5, 15, 3):  
>     print(x)  
5  
8  
11  
14
```

Question – What is printed ?

Python shell

```
> for i in range(1, 4):
>     for j in range(i, 4):
>         print(i, j, sep=':', end=' ')
```

- a) 1:1 1:2 1:3 2:1 2:2 2:3 3:1 3:2 3:3
- b) 1:1 1:2 1:3 2:2 2:3 3:3
- c) 1:1 2:1 3:1 1:2 2:2 3:2 1:3 2:3 3:3
- d) 1:1 2:1 3:1 2:2 3:2 3:3
- e) Don't know

Question – **break**, what is printed ?

Python shell

```
> for i in range(1, 4):
>     for j in range(1, 4):
>         print(i, j, sep=':', end=' ')
>         if j >= i:
>             break
```

- a) *nothing*
- b) 1:1
- c) 1:1 2:1 2:2 3:1 3:2 3:3
- d) 1:1 2:2 3:3
- e) Don't know



In nested `for`- and `while`-loops,
`break` only breaks the innermost loop

```
***** COMMODORE 64 BASIC V2 *****  
64K RAM SYSTEM 38911 BASIC BYTES FREE  
READY.  
10 FOR I=1 TO 10  
20 PRINT I  
30 NEXT  
RUN  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
READY.
```

```
***** COMMODORE 64 BASIC V2 *****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
10 FOR I=1 TO 3
20   FOR J=I TO 3
30     PRINT I,J
40   NEXT
50 NEXT
RUN
1
112233
123233
READY.
```

Palindromic substrings

- Find all **palindromic** substrings of length ≥ 2 , i.e. substrings spelled identically forward and backwards:

abraca**d**rabra**t**ra**ll**alla
 i j i j

- Algorithm:** Test all possible substrings (brute force/exhaustive search)
- Note:** the slice `t [::-1]` is `t` reversed

palindrom.py

```
s = 'abracadabrabratrallalla'

for i in range(len(s)):
    for j in range(i + 2, len(s) + 1):
        t = s[i:j]
        if t == t[::-1]:
            print(t)
```

Python shell

```
aca
alla
allalla
ll
llall
lal
alla
ll
```

Sieve of Eratosthenes

- Find all prime numbers $\leq n$
- Algorithm:

2 3 4 5 6 7 8 9 10 11 12 13 14 ...
2 3 4 5 6 7 8 9 10 11 12 13 14 ...
2 3 4 5 6 7 8 9 10 11 12 13 14 ...
2 3 4 5 6 7 8 9 10 11 12 13 14 ...
2 3 4 5 6 7 8 9 10 11 12 13 14 ...
2 3 4 5 6 7 8 9 10 11 12 13 14 ...

eratosthenes.py

```
n = 100
prime = [True] * (n + 1)

for i in range(2, n):
    for j in range(2 * i, n + 1, i):
        prime[j] = False

for i in range(2, n + 1):
    if prime[i]:
        print(i, end=' ')
```

Python shell

```
| 2 3 5 7 11 13 17 19 23 29 31 37 41
| 43 47 53 59 61 67 71 73 79 83 89
| 97
```

while-else and for-else loops

- Both for- and while-loops can have an optional “else”:

```
for variable in sequence:  
    block  
else:  
    block
```

```
while condition:  
    block  
else:  
    block
```

- The “else” block is only executed if no `break` is performed in the loop
- The “else” construction for loops is specific to Python, and does not exist in e.g. C, C++ and Java



Linear search

linear-search-while.py

```
L = [7, 3, 6, 4, 12, 'a', 8, 13]
x = 4

i = 0
while i < len(L):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        break
    i = i + 1

if i >= len(L):
    print(x, 'not in', L)
```

linear-search-while-else.py

```
i = 0
while i < len(L):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        break
    i = i + 1
else:
    print(x, 'not in', L)
```

linear-search-for.py

```
found = False
for i in range(len(L)):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        found = True
        break

if not found:
    print(x, 'not in', L)
```

linear-search-for-else.py

```
for i in range(len(L)):
    if L[i] == x:
        print(x, 'at position', i, 'in', L)
        break
else:
    print(x, 'not in', L)
```

linear-search-builtin.py

```
if x in L:
    print(x, 'at position', L.index(x), 'in', L)
else:
    print(x, 'not in', L)
```

Some performance considerations

String concatenation

- To concatenate two (or few) strings use

$str_1 + str_2$
 $var += str$

- To concatenate several/many strings use

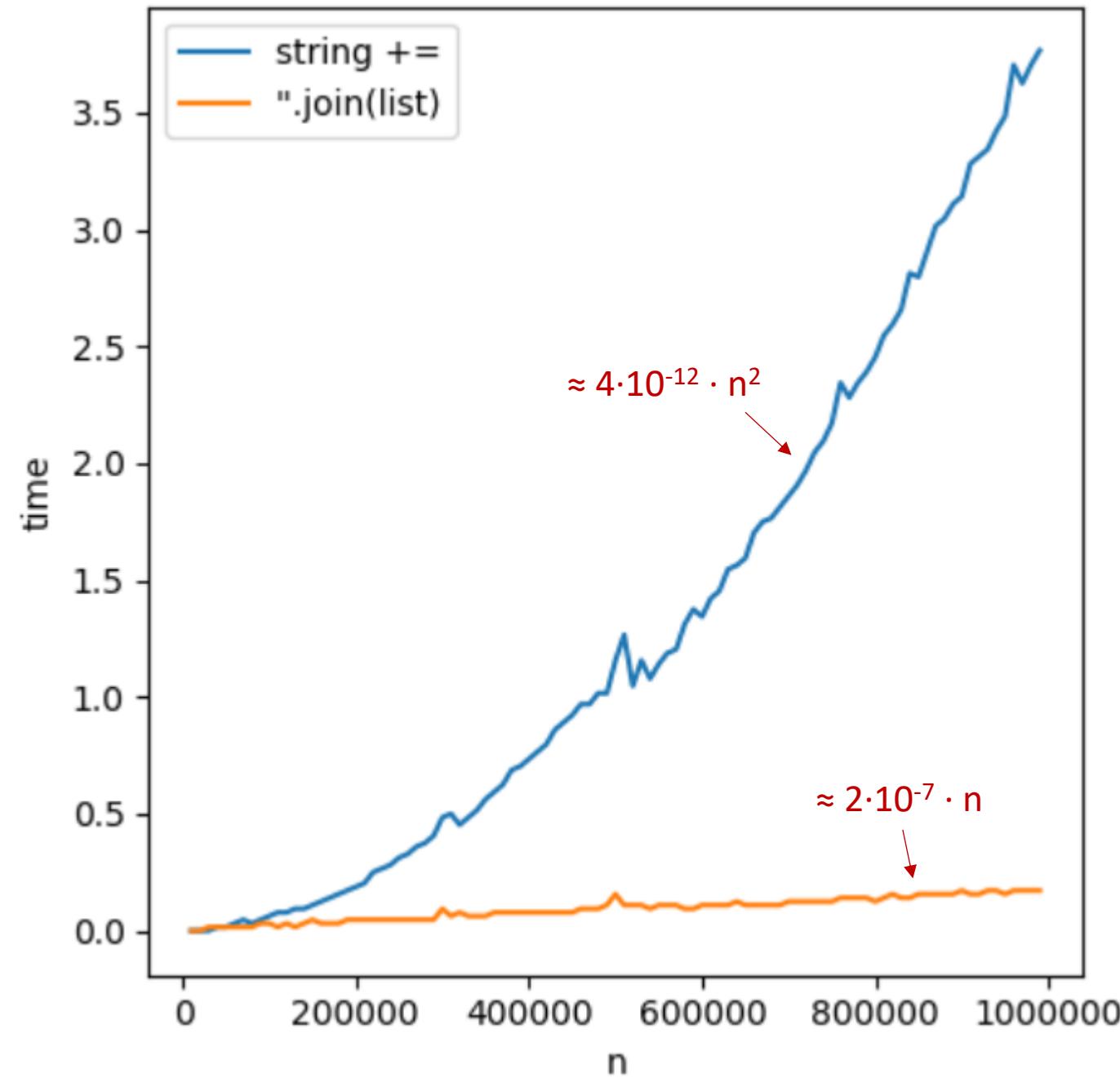
`' '.join([str1, str2, str3, ... , strn])`

- Concatenating several strings by repeated use of `+` generates explicitly the longer-and-longer intermediate results; using `join` avoids this slowdown



Python shell

```
> s = 'A' + 'B' + 'C'  
> s  
| 'ABC'  
> 'x'.join(['A', 'B', 'C'])  
| 'AxBxC'  
> s = ''  
> s += 'A'  
> s += 'B'  
> s += 'C'  
> s  
| 'ABC'  
> L = []  
> L.append('A')  
> L.append('B')  
> L.append('C')  
> L  
| ['A', 'B', 'C']  
> s = ' '.join(L)  
> s  
| 'ABC'
```



string-concatenation.py

```
from time import time
from matplotlib import pyplot as plt

ns = range(10_000, 1_000_000, 10_000)
time_string = []
time_list = []
for n in ns:
    start = time()
    s = ''
    for _ in range(n):
        s += 'abcdefg' # slow
    end = time()
    time_string.append(end - start)

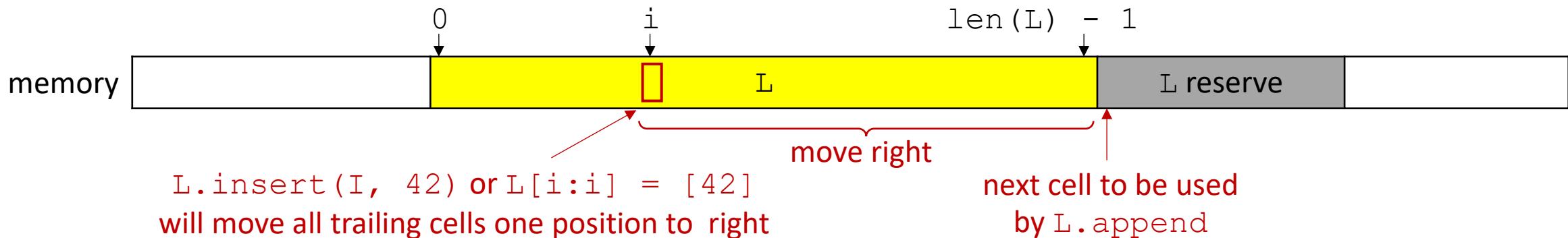
    start = time()
    substrings = []
    for _ in range(n):
        substrings.append('abcdefg');
    s = ''.join(substrings);
    end = time()
    time_list.append(end - start)

plt.plot(ns, time_string, label='string +=')
plt.plot(ns, time_list, label=".join(list)")
plt.xlabel('n')
plt.ylabel('time')
plt.legend()
plt.show()
```

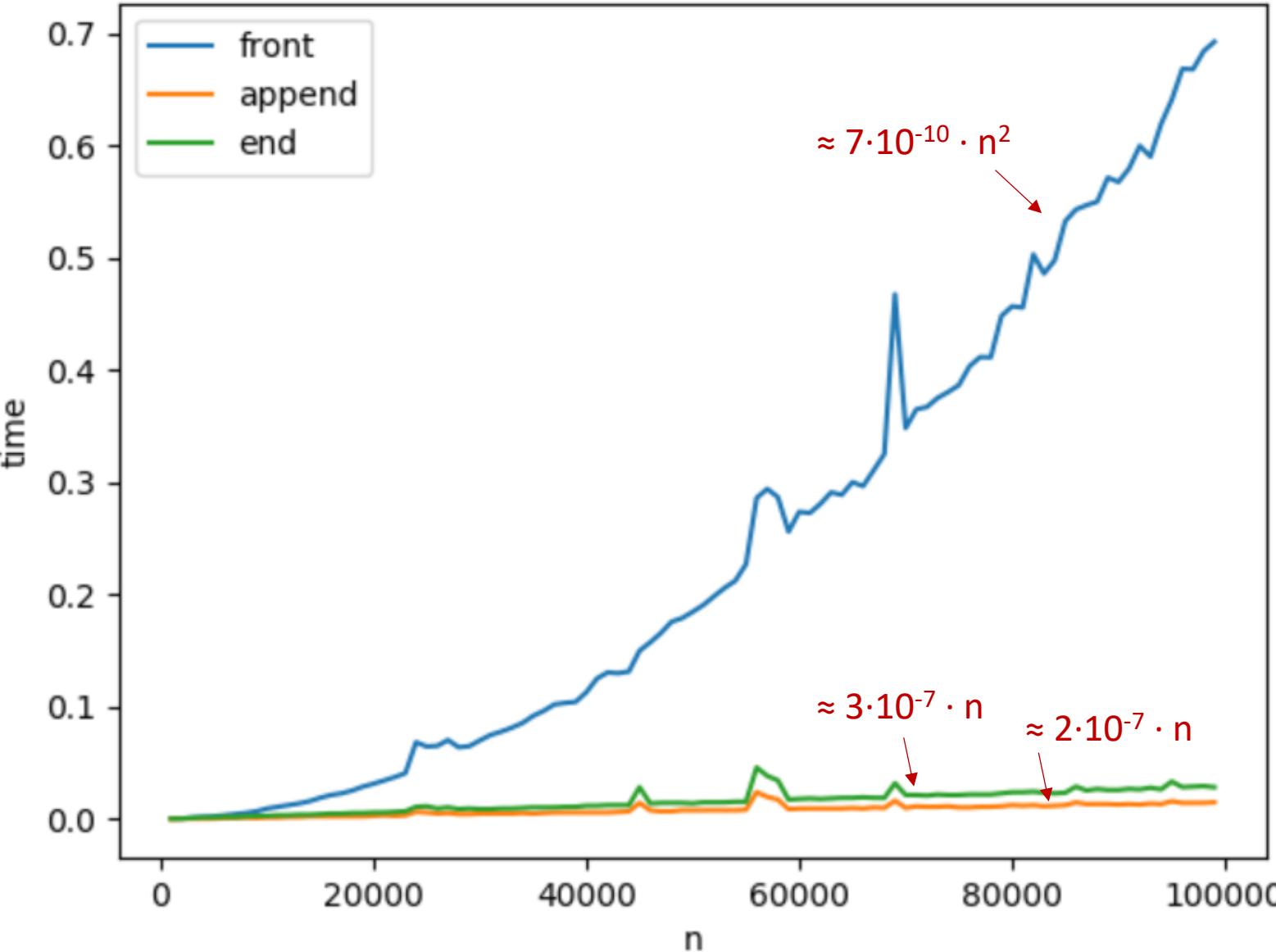


The internal implementation of Python lists

- Accessing and updating list positions take the same time independently of position
- Creating new / deleting entries in a list depends on position,
Python optimizes towards updates at the end
- Try to organize your usage of lists to insert / delete elements at the end
`L.append(element)` and `L.pop()`
- Python lists internally have space for adding $\approx 12.5\%$ additional entries at the end;
when the reserved extra space is exhausted the list is moved to a new chunk of
memory with $\approx 12.5\%$ extra space



List insertions at front vs end



```
list-insertions.py
from time import time
from matplotlib import pyplot as plt
ns = range(1000, 100_000, 1000)
time_end = []
time_append = []
time_front = []
for n in ns:
    start = time()
    L = []
    for i in range(n):
        L[i:i] = [i] # insert after list
    end = time()
    time_end.append(end - start)

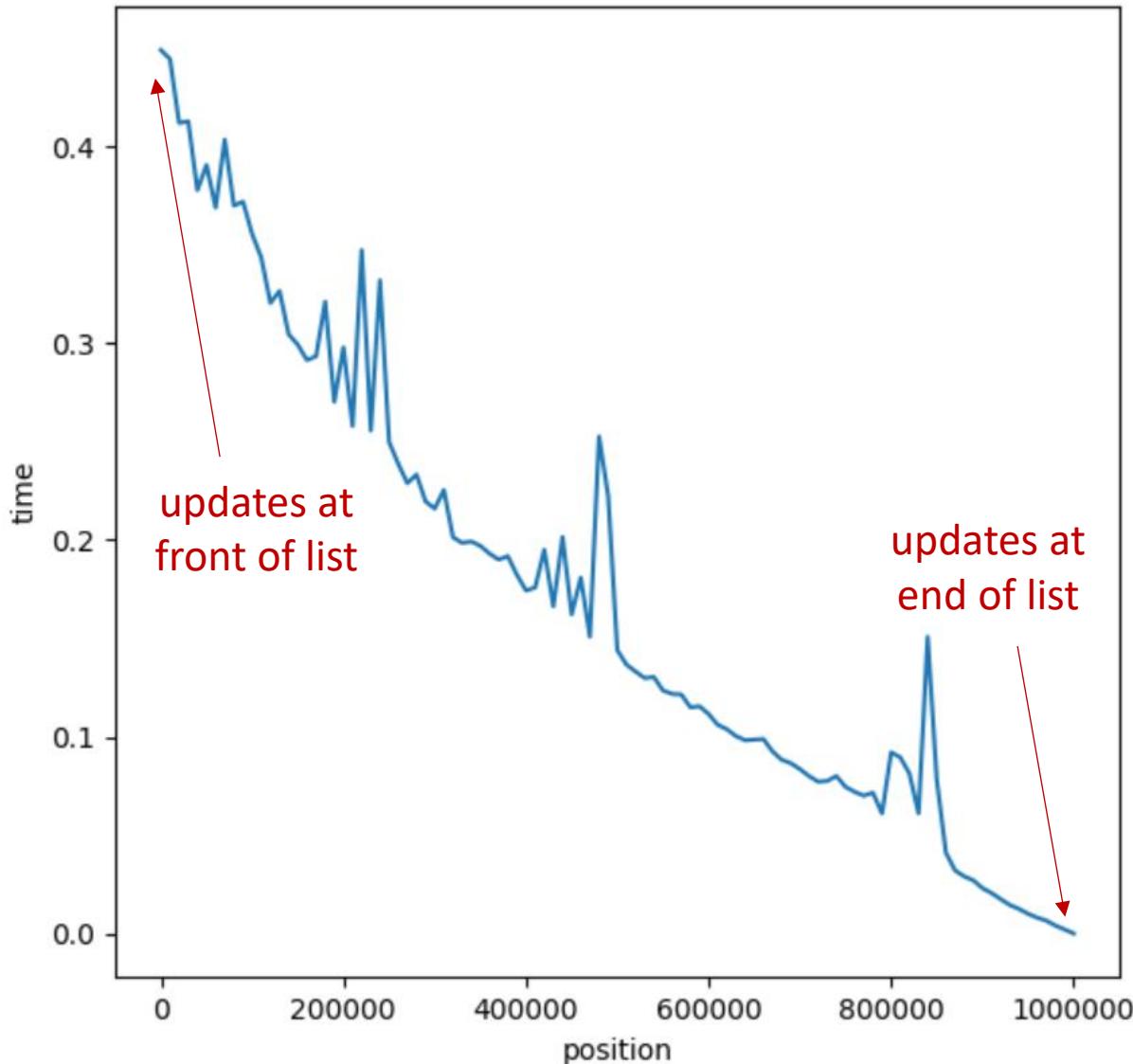
    start = time()
    L = []
    for i in range(n):
        L.append(i) # append to list
    end = time()
    time_append.append(end - start)

    start = time()
    L = []
    for i in range(n):
        L[0:0] = [i] # insert at front
    end = time()
    time_front.append(end - start)

plt.plot(ns, time_front, label='front')
plt.plot(ns, time_append, label='append')
plt.plot(ns, time_end, label='end')
plt.xlabel('n')
plt.ylabel('time')
plt.legend()
plt.show()
```

⚠️

Updates (insertions + deletions) in the middle of a list



list-updates.py

```
from time import time
from matplotlib import pyplot as plt

ns = range(0, 1_000_001, 10_000)
time_pos = []
L = list(range(1_000_000)) # L = [0, ..., 999_999]
for i in ns:
    start = time()
    for _ in range(1000):
        L[i:i] = [42] # insert element before L[i]
        del L[i]       # remove L[i] from L
    end = time()
    time_pos.append(end - start)

plt.plot(ns, time_pos)
plt.xlabel('position')
plt.ylabel('time')
plt.show()
```