Lecture 5: Lists

INF 605 · 2026-09-10

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. Last class was **Loops and Iteration**. Name one thing we actually ran or wrote (`while`, `for`, or `range`).
2. What does `range(5)` produce — does it include 5?

5.2 Lists

- Many of the capabilities shown in this section apply to all sequence types.

Creating a List

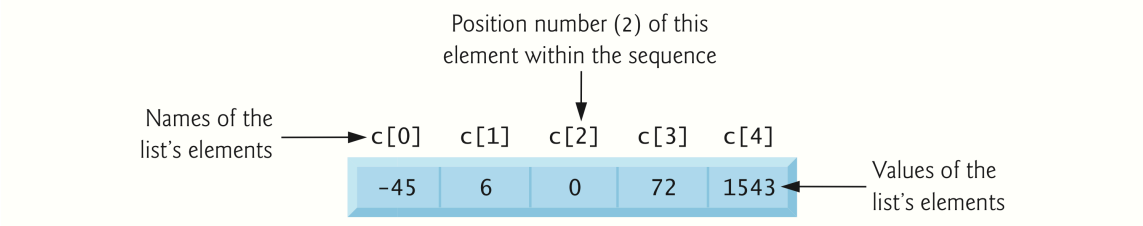
- **Lists** typically store **homogeneous data**, but may store **heterogeneous data**.

In [1]:
```
c = [-45, 6, 0, 72, 1543]
c
```

Out[1]:
```
[-45, 6, 0, 72, 1543]
```

Accessing Elements of a List

- Reference a list element by writing the list's name followed by the element's **index** enclosed in `[]` (the **subscription operator**).



In [2]:
```
c[0]
```

```
Out[2]: -45
```

The value at index `4` is the last element in this five-element list.

```
In [3]: c[4]
```

```
Out[3]: 1543
```

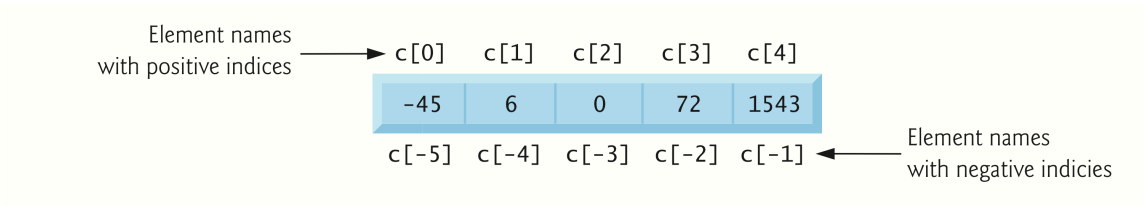
Determining a List's Length

```
In [4]: len(c)
```

```
Out[4]: 5
```

Accessing Elements from the End of the List with Negative Indices

- Lists can be accessed from the end by using *negative indices*:




```
In [5]: c[-1]
```

```
Out[5]: 1543
```

```
In [6]: c[-5]
```

```
Out[6]: -45
```

Lists Are Mutable

```
In [7]: c[4]
```

```
Out[7]: 1543
```

Replace the element at index `4`.

```
In [8]: c[4] = 17
```

```
In [9]: c
```

```
Out[9]: [-45, 6, 0, 72, 17]
```

Some Sequences Are Immutable

- Python's string sequences are immutable.

In [10]:
```
s = 'hello'
```

In [11]:
```
s[0]
```

Out[11]:
```
'h'
```

In [12]:
```
s[0] = 'H'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 s[0] = 'H'

TypeError: 'str' object does not support item assignment
```

Using List Elements in Expressions

In [13]:
```
c = [-45, 6, 0, 72, 1543]
c[0] + c[1] + c[2]
```

Out[13]:
```
-39
```

Appending to a List with +=

- Lists can grow dynamically to accommodate new items.

In [14]:
```
a_list = []
```

Each pass appends one integer, wrapped in a one-element list.

In [15]:
```
for number in range(1, 6):
    a_list += [number]
a_list
```

Out[15]:
```
[1, 2, 3, 4, 5]
```

A string is a sequence of characters, so `+=` appends each character.

In [16]:
```
letters = []
letters += 'Python'
letters
```

Out[16]:
```
['P', 'y', 't', 'h', 'o', 'n']
```

A one-element list appends the whole string as a single item.

In [17]:
```
letters += ['Python']
letters
```

Out[17]:
```
['P', 'y', 't', 'h', 'o', 'n', 'Python']
```

Concatenating Lists with +

- Can **concatenate** two lists using `+` to create a *new* list.

In [18]:
```
list1 = [10, 20, 30]
```

In [19]:
```
list2 = [40, 50]
```

In [20]:
```
concatenated_list = list1 + list2
```

In [21]:
```
concatenated_list
```

Out[21]:
```
[10, 20, 30, 40, 50]
```

Using `for` and `range` to Access List Indices and Values

In [22]:
```
for i in range(len(concatenated_list)):
    print(f'index{i}: {concatenated_list[i]}')
```

```
index0: 10
index1: 20
index2: 30
index3: 40
index4: 50
```

What's wrong with this code

Index `10` is past the end of a five-element list.

In [23]:
```
numbers = [1, 2, 3, 4, 5]
numbers[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[23], line 2
      1 numbers = [1, 2, 3, 4, 5]
----> 2 numbers[10]

IndexError: list index out of range
```

You can read a character from a string, but you cannot assign to one.

In [24]:
```
name = 'Amanda'
name[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 2
      1 name = 'Amanda'
----> 2 name[0] = 'A'

TypeError: 'str' object does not support item assignment
```

List indices must be integers (or slices), not floats.

In [25]:
```
numbers = [1, 2, 3, 4, 5]
numbers[3.2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 2
      1 numbers = [1, 2, 3, 4, 5]
----> 2 numbers[3.2]

TypeError: list indices must be integers or slices, not float
```

Exercise (index)

`temps` holds five noon temperatures. Fill in the four assignments.

In [ ]:
```
temps = [68, 71, 74, 69, 73]
first = None      # temps[0]
last = None       # last element, using a negative index
n_days = None     # len(...)
temps[2] = None   # replace Wednesday with 80
print(first, last, n_days, temps)
```

Solution

In [26]:
```
temps = [68, 71, 74, 69, 73]
first = temps[0]
last = temps[-1]
n_days = len(temps)
temps[2] = 80
print(first, last, n_days, temps)
```

```
68 73 5 [68, 71, 80, 69, 73]
```

5.5 Sequence Slicing

- Can **slice** sequences to create new sequences of the same type containing *subsets* of the original elements.
- Slice operations that do *not* modify a sequence work the same way for lists and strings.

Specifying a Slice with Starting and Ending Indices

In [27]:
```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
# index     0  1  2  3   4   5   6   7
numbers[2:6]  # index 6 is not included, same idea as range()
```

Out[27]:
```
[5, 7, 11, 13]
```

Specifying a Slice with Only an Ending Index

- Starting index `0` is assumed.

In [28]:
```
numbers[:6]
```

Out[28]:
```
[2, 3, 5, 7, 11, 13]
```

In [29]:
```
numbers[0:6]
```

Out[29]:
```
[2, 3, 5, 7, 11, 13]
```

Specifying a Slice with Only a Starting Index

- Assumes the sequence's length as the ending index.

In [30]:
```
numbers[6:]
```

Out[30]:
```
[17, 19]
```

In [31]:
```
numbers[6:len(numbers)]
```

Out[31]:
```
[17, 19]
```

Specifying a Slice with No Indices

In [32]:
```
numbers[:]
```

Out[32]:
```
[2, 3, 5, 7, 11, 13, 17, 19]
```

- Slices create new lists. The new list's elements refer to the *same objects* as the original list's elements (a **shallow copy**).

Slicing with Steps

In [33]:
```
numbers[::2]
```

Out[33]:
```
[2, 5, 11, 17]
```

Slicing with Negative Indices and Steps

In [34]:
```
numbers[::-1]
```

Out[34]:
```
[19, 17, 13, 11, 7, 5, 3, 2]
```

In [35]:
```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
# index     0  1  2  3   4   5   6   7
#          -8 -7 -6 -5  -4  -3  -2  -1
numbers[-1:-9:-1]
```

Out[35]:
```
[19, 17, 13, 11, 7, 5, 3, 2]
```

Modifying Lists Via Slices

- Can modify a list by assigning to a slice.

In [36]:
```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
numbers[0:3] = ['two', 'three', 'five']
numbers
```

Out[36]:
```
['two', 'three', 'five', 7, 11, 13, 17, 19]
```

Assigning an empty list to a slice deletes those elements.

In [37]:
```
numbers[0:3] = []
numbers
```

Out[37]:
```
[7, 11, 13, 17, 19]
```

A step on the left-hand side replaces every other element.

In [38]:
```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
numbers[::2] = [100, 100, 100, 100]
numbers
```

Out[38]:
```
[100, 3, 100, 7, 100, 13, 100, 19]
```

`numbers[:] = []` empties the **same** list object. `numbers = []` names a **new** empty list.

In [39]:
```
numbers[:] = []
numbers
```

Out[39]:
```
[]
```

Exercise (slice)

Using `numbers` below, fill in each slice.

In [ ]:
```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
first_three = None      # numbers[0:3]
from_eleven = None      # 11 through the end
every_other = None      # step 2
backwards = None        # reverse the whole list
print(first_three, from_eleven, every_other, backwards)
```

Solution

In [40]:
```
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
first_three = numbers[0:3]
from_eleven = numbers[4:]
every_other = numbers[::2]
backwards = numbers[::-1]
print(first_three, from_eleven, every_other, backwards)
```

```
[2, 3, 5] [11, 13, 17, 19] [2, 5, 11, 17] [19, 17, 13, 11, 7, 5, 3, 2]
```

5.6 del Statement

Deleting the Element at a Specific List Index

In [41]:
```
numbers = list(range(0, 10))
```

In [42]:
```
numbers
```

Out[42]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`del numbers[-1]` removes the last element.

In [43]:
```
del numbers[-1]
```

In [44]:
```
numbers
```

Out[44]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Deleting a Slice from a List

In [45]:
```
del numbers[0:2]
```

In [46]:
```
numbers
```

Out[46]:
```
[2, 3, 4, 5, 6, 7, 8]
```

`del numbers[::2]` deletes every other remaining element.

In [47]:
```
del numbers[::2]
```

In [48]:
```
numbers
```

Out[48]:
```
[3, 5, 7]
```

Deleting a Slice Representing the Entire List

In [49]:
```
del numbers[:]
```

In [50]:
```
numbers
```

Out[50]:
```
[]
```

Deleting a Variable from the Current Session

- After `del numbers`, the name `numbers` no longer exists.

In [51]:
```
del numbers
```

In [52]:
```
numbers
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[52], line 1
----> 1 numbers

NameError: name 'numbers' is not defined
```

5.8 Sorting Lists

Sorting a List in Ascending Order

- List method `sort` *modifies* a list.

In [53]:
```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

In [54]:
```
numbers.sort()
```

In [55]:
```
numbers
```

Out[55]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting a List in Descending Order

In [56]:
```
numbers.sort(reverse=True)
```

In [57]:
```
numbers
```

Out[57]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Built-In Function `sorted`

- Built-in function **`sorted`** *returns a new list* containing the sorted elements of its argument sequence — the original sequence is *unmodified*.

In [58]:
```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

In [59]:
```
ascending_numbers = sorted(numbers)
```

In [60]:
```
ascending_numbers
```

Out[60]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`numbers` itself is unchanged.

In [61]:
```
numbers
```

Out[61]:
```
[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

`sorted` also works on a string: it returns a list of characters.

In [62]:
```
letters = 'fadgchjebi'
ascending_letters = sorted(letters)
ascending_letters
```

Out[62]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

In [63]:
```
letters
```

Out[63]:
```
'fadgchjebi'
```

Exercise (sort)

Sort a copy. Leave `scores` in its original order.

In [ ]:
```
scores = [88, 71, 99, 64, 92]
low_to_high = None   # sorted(scores)
high_to_low = None   # sorted(scores, reverse=True)
print(low_to_high)
print(high_to_low)
print(scores)        # should still be [88, 71, 99, 64, 92]
```

Solution

In [64]:
```
scores = [88, 71, 99, 64, 92]
low_to_high = sorted(scores)
```

```
high_to_low = sorted(scores, reverse=True)
print(low_to_high)
print(high_to_low)
print(scores)
```

```
[64, 71, 88, 92, 99]
[99, 92, 88, 71, 64]
[88, 71, 99, 64, 92]
```

5.9 Searching Sequences

- **Searching** is the process of locating a particular **key** value.

List Method index

- Searches through a list from index 0 and returns the index of the *first* element that matches the search key.
- `ValueError` if the value is not in the list.

In [65]:
```
numbers = [3, 7, 1, 5, 2, 8, 5, 6]
```

In [66]:
```
numbers.index(5)
```

Out[66]: 3

Specifying the Starting Index of a Search

In [67]:
```
numbers *= 2
```

In [68]:
```
numbers
```

Out[68]: [3, 7, 1, 5, 2, 8, 5, 6, 3, 7, 1, 5, 2, 8, 5, 6]

`numbers.index(5, 7)` starts looking at index 7.

In [69]:
```
numbers.index(5, 7)
```

Out[69]: 11

Specifying the Starting and Ending Indices of a Search

- Look for `7` among the elements with indices `5` through `9`.

In [70]:
```
numbers.index(7, 5, 10)
```

Out[70]: 9

Operators `in` and `not in`

- Operator `in` tests whether its right operand's iterable contains the left operand's value.

In [71]:
```
numbers = [3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
if 3 in numbers:
    print('The number is in the list')
else:
    print('The number is not in the list')
```

```
The number is in the list
```

In [72]:
```
5 in numbers
```

Out[72]:
```
True
```

- Operator `not in` tests the opposite.

In [73]:
```
1000 not in numbers
```

Out[73]:
```
True
```

In [74]:
```
5 not in numbers
```

Out[74]:
```
False
```

Using Operator `in` to Prevent a `ValueError`

- Call `.index` only after you know the key is present.

In [75]:
```
key = 1000
if key in numbers:
    print(f'found {key} at index {numbers.index(key)}')
else:
    print(f'{key} not found')
```

```
1000 not found
```

Exercise (search)

Find the first `5`, then test membership.

In [ ]:
```
nums = [3, 7, 1, 5, 2, 8, 5, 6]
where = None          # nums.index(5)
has_nine = None       # 9 in nums
missing_ok = None     # 9 not in nums
print(where, has_nine, missing_ok)
```

Solution

In [76]:
```
nums = [3, 7, 1, 5, 2, 8, 5, 6]
where = nums.index(5)
has_nine = 9 in nums
missing_ok = 9 not in nums
print(where, has_nine, missing_ok)
```

```
3 False True
```

5.10 Other List Methods

In [77]:
```
color_names = ['orange', 'yellow', 'green']
```

Inserting an Element at a Specific List Index

In [78]:
```
color_names.insert(0, 'red')
```

In [79]:
```
color_names
```

Out[79]:
```
['red', 'orange', 'yellow', 'green']
```

Adding an Element to the End of a List

In [80]:
```
color_names.append('blue')
```

In [81]:
```
color_names
```

Out[81]:
```
['red', 'orange', 'yellow', 'green', 'blue']
```

Adding All the Elements of a Sequence to the End of a List

- Equivalent to `+=`.

In [82]:
```
color_names.extend(['indigo', 'violet'])
```

In [83]:
```
color_names
```

Out[83]:
```
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

`extend` walks any sequence. A string contributes one character at a time.

In [84]:
```
sample_list = []
sample_list.extend('abc')
sample_list
```

Out[84]:
```
['a', 'b', 'c']
```

Removing the First Occurrence of an Element in a List

- `ValueError` occurs if `remove`'s argument is not in the list.

```
In [85]: color_names.remove('green')
```

```
In [86]: color_names
```

```
Out[86]: ['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Emptying a List

```
In [87]: color_names.clear()
```

```
In [88]: color_names
```

```
Out[88]: []
```

Counting the Number of Occurrences of an Item

```
In [89]: responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3,
                      1, 4, 3, 3, 3, 2, 3, 3, 2, 2]
         for i in range(1, 6):
             print(f'{i} appears {responses.count(i)} times in responses')
```

```
1 appears 3 times in responses
2 appears 5 times in responses
3 appears 8 times in responses
4 appears 2 times in responses
5 appears 2 times in responses
```

Reversing a List's Elements

- Method **`reverse`** reverses the contents of a list in place.

```
In [90]: color_names = ['red', 'orange', 'yellow', 'green', 'blue']
```

```
In [91]: color_names.reverse()
```

```
In [92]: color_names
```

```
Out[92]: ['blue', 'green', 'yellow', 'orange', 'red']
```

Copying a List

- Method `copy` returns a *new* list containing a *shallow* copy.

```
In [93]: copied_list = color_names.copy()
```

```
In [94]: copied_list
```

Out[94]:
```
['blue', 'green', 'yellow', 'orange', 'red']
```

Exercise (methods)

Build the rainbow list with methods, then copy it.

In [ ]:
```
colors = ['yellow']
# insert 'red' at index 0
# append 'blue'
# extend with ['indigo', 'violet']
n_yellow = None     # colors.count('yellow')
snapshot = None     # colors.copy()
print(colors)
print(n_yellow, snapshot)
```

Solution

In [95]:
```
colors = ['yellow']
colors.insert(0, 'red')
colors.append('blue')
colors.extend(['indigo', 'violet'])
n_yellow = colors.count('yellow')
snapshot = colors.copy()
print(colors)
print(n_yellow, snapshot)
```

```
['red', 'yellow', 'blue', 'indigo', 'violet']
1 ['red', 'yellow', 'blue', 'indigo', 'violet']
```

5.11 Simulating Stacks with Lists

- Python does not have a built-in stack type.
- Can think of a stack as a constrained list.
- *Push* using list method `append`.
- *Pop* using list method **`pop`** with no arguments to get items in last-in, first-out (LIFO) order.

In [96]:
```
stack = []
```

In [97]:
```
stack.append('red')
```

In [98]:
```
stack
```

Out[98]:
```
['red']
```

`append` again pushes a second page on top.

```
In [99]: stack.append('green')
```

```
In [100…: stack
Out[100…: ['red', 'green']
```

`pop()` returns the top item and removes it.

```
In [101…: stack.pop()
Out[101…: 'green'
```

```
In [102…: stack
Out[102…: ['red']
```

```
In [103…: stack.pop()
Out[103…: 'red'
```

```
In [104…: stack
Out[104…: []
```

Popping an empty stack raises `IndexError`.

```
In [105…: try:
              stack.pop()
          except IndexError:
              print('cannot pop an empty stack')
cannot pop an empty stack
```

- You can also use a list to simulate a **queue**.
- Items are retrieved from queues in **first-in, first-out (FIFO) order**.

5.16 Two-Dimensional Lists

- Lists can contain other lists as elements.
- Typical use is to represent **tables** of values arranged in **rows** and **columns**.
- To identify a particular table element, we specify *two* indices — the first identifies the element's row, the second the element's column.

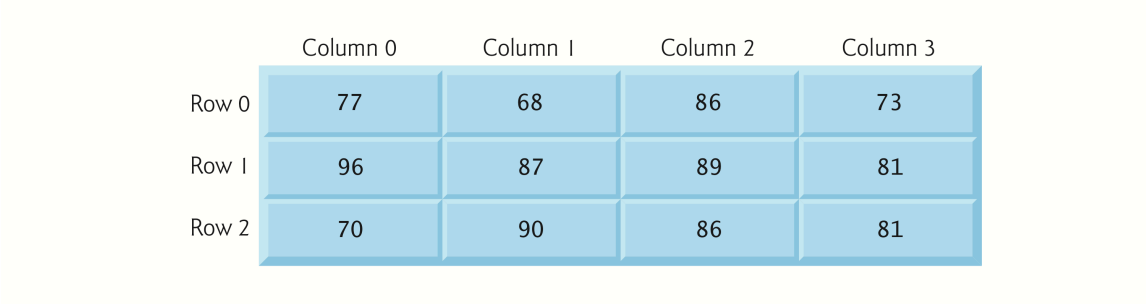
Creating a Two-Dimensional List

```
In [106…: a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]
```

Writing the list as follows makes its row and column tabular structure clearer:

```
a = [[77, 68, 86, 73],  # first student's grades
     [96, 87, 89, 81],  # second student's grades
     [70, 90, 86, 81]]  # third student's grades
```

Illustrating a Two-Dimensional List

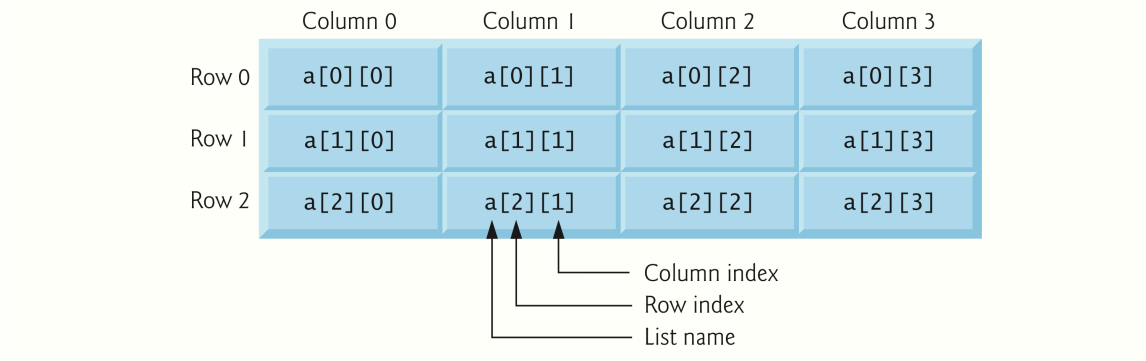


Row index `2`, column index `3` is the last exam of the third student.

In [107…
```
a[2][3]
```

Out[107… 81

Identifying the Elements in a Two-Dimensional List



- Output the rows of the two-dimensional list.

In [108…
```
for row in a:
    for item in row:
        print(item, end=' ')
    print()
```

```
77 68 86 73
96 87 89 81
70 90 86 81
```

How the Nested Loops Execute

- The outer `for` statement iterates over the list's rows one row at a time.

- During each iteration of the outer `for`, the inner `for` iterates over *each* column in the current row.

In [109…
```
for i, row in enumerate(a):
    for j, item in enumerate(row):
        print(f'a[{i}][{j}]={item} ', end=' ')
    print()
```

```
a[0][0]=77  a[0][1]=68  a[0][2]=86  a[0][3]=73
a[1][0]=96  a[1][1]=87  a[1][2]=89  a[1][3]=81
a[2][0]=70  a[2][1]=90  a[2][2]=86  a[2][3]=81
```

Exercise (2d)

Read one cell, then print every grade.

In [ ]:
```
table = [
    [77, 68, 86, 73],
    [96, 87, 89, 81],
    [70, 90, 86, 81],
]
first_exam = None   # table[0][0]
last_exam = None    # table[2][3]
print(first_exam, last_exam)
# nested for: print each row on its own line
```

Solution

In [110…
```
table = [
    [77, 68, 86, 73],
    [96, 87, 89, 81],
    [70, 90, 86, 81],
]
first_exam = table[0][0]
last_exam = table[2][3]
print(first_exam, last_exam)
for row in table:
    for item in row:
        print(item, end=' ')
    print()
```

```
77 81
77 68 86 73
96 87 89 81
70 90 86 81
```

List comprehensions (short map)

A loop that builds a list:

```
squares = []
for n in range(1, 6):
    squares.append(n * n)
```

is the same idea as:

```
squares = [n * n for n in range(1, 6)]
```

That one-line form is a **list comprehension**. We will practice mapping and filtering on **Lecture 9**.

In [111…
```
squares = []
for n in range(1, 6):
    squares.append(n * n)
squares
```

Out[111… `[1, 4, 9, 16, 25]`

The comprehension version:

In [112…
```
[n * n for n in range(1, 6)]
```

Out[112… `[1, 4, 9, 16, 25]`

Comprehensive exercises

Comprehensive exercise 1

A cart starts empty. Add three items, remove one, then report what is left and a sorted copy.

In [ ]:
```
cart = []
# append 'milk', 'eggs', 'bread'
# remove 'eggs'
left = None          # cart itself after the remove
alpha = None         # sorted(cart)
print(left)
print(alpha)
```

Solution

In [113…
```
cart = []
cart.append('milk')
cart.append('eggs')
cart.append('bread')
cart.remove('eggs')
left = cart
alpha = sorted(cart)
print(left)
print(alpha)
```

```
['milk', 'bread']
['bread', 'milk']
```

Comprehensive exercise 2

From the grade table, collect the last exam in every row (the last column), then take the highest of those three scores. Use a loop, not a comprehension.

In [ ]:
```python
grades = [
    [77, 68, 86, 73],
    [96, 87, 89, 81],
    [70, 90, 86, 81],
]
last_column = []     # append grades[i][-1] for each row
highest = None       # max(last_column)
print(last_column, highest)
```

Solution

In [114…
```python
grades = [
    [77, 68, 86, 73],
    [96, 87, 89, 81],
    [70, 90, 86, 81],
]
last_column = []
for row in grades:
    last_column.append(row[-1])
highest = max(last_column)
print(last_column, highest)
```

```
[73, 81, 81] 81
```

Takeaways

- A list is a mutable sequence. Index `0` is the first element; `-1` is the last.
- Slices `start:stop:step` make a new list. The stop index is not included.
- `del`, `append`, `insert`, `remove`, `pop`, `extend`, `clear`, `reverse`, `copy`, and `sort` change a list in place. `sorted` and `+` build a new list.
- `index` / `in` / `count` search. Check `in` before `index` if the key might be missing.
- `append` / `pop()` at the end is a stack (last in, first out).
- A list of lists is a table: `a[row][column]`. Nested `for` walks every cell.
- Next class: **Functions**.