CSCI 145 · Introduction to Computing · Fall 2026

Sets and References

Lecture 13 — collections without duplicates, and what a variable holds

Outline

- Sets
- References
- Lists of lists and copies

Part 1 of 3

Sets

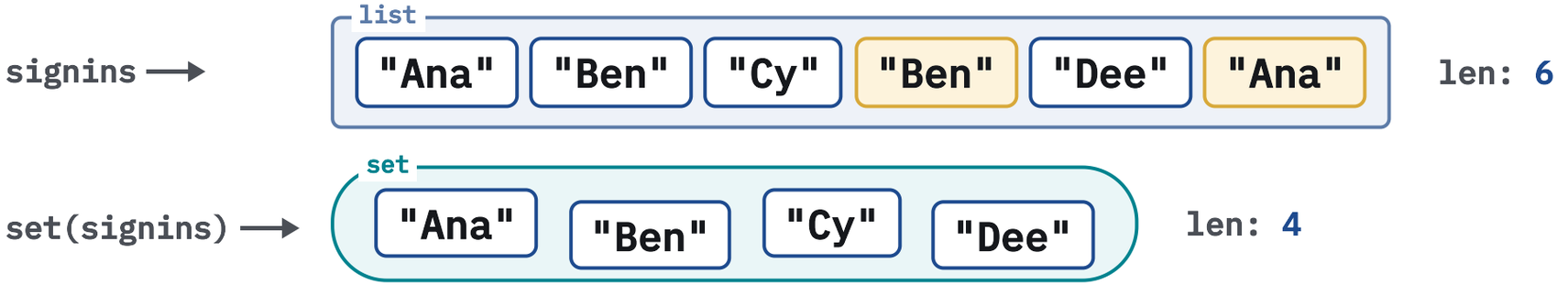
> A **set** is a collection with no duplicates and no order
> like the names of the people in this room

Who's here?

Everyone signs in on the way in

Some people sign in twice

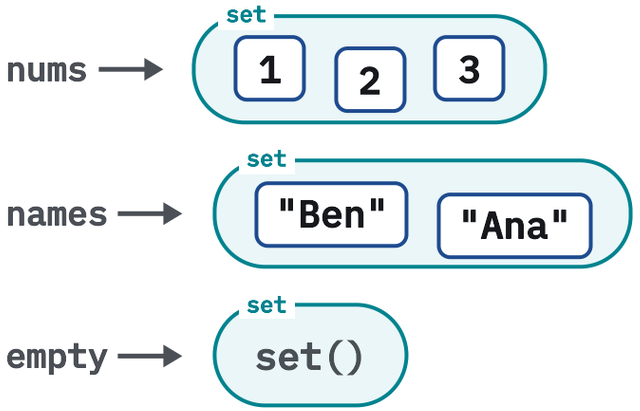
```
>>> signins.append("    ")
                    -----
```



- A list keeps every sign-in, in order
- `set(signins)` keeps each name once
  Its length is the number of people here

Making a set

```
>>> nums = {3, 1, 3, 2}
>>> nums
{1, 2, 3}
>>> len(nums)
3
>>> 2 in nums
True
>>> names = set(["Ana", "Ben", "Ana"])
>>> names
{'Ben', 'Ana'}
>>> type({})
<class 'dict'>
>>> empty = set()
>>> len(empty)
0
```



- Curly braces, like a dictionary but no colons
- Duplicates disappear
- `set(...)` makes a set from a list
- No order: a set may print its values in any order
- `{}` is an empty dictionary
  `set()` is an empty set

Changing a set

```
>>> here = {"Ana", "Ben"}
>>> here.add("Cy")
>>> here.add("Ana")
>>> here.remove("Ben")
>>> here
{'Ana', 'Cy'}
>>> here.remove("Zoe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Zoe'
>>> here.discard("Zoe")
>>> here[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

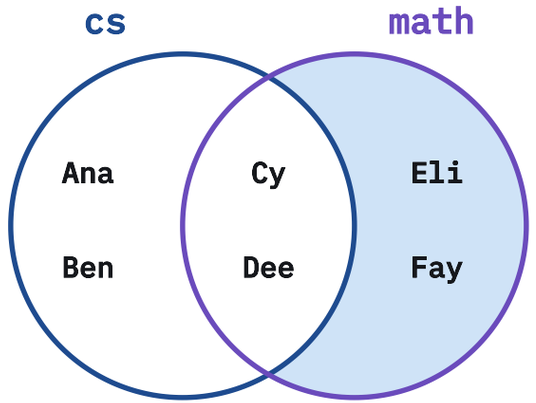


- `.add` puts a value in
- Already there? Nothing happens
- `.remove` takes a value out
  Not there? `KeyError`
- `.discard` skips the error
- No order, so no indexing
- Like dictionary keys, the values in a set must be immutable

Who's in both classes?

The students in CS 145 and in Math 121

```
>>> cs = {"Ana", "Ben", "Cy", "Dee"}
>>> math = {"Cy", "Dee", "Eli", "Fay"}
>>> cs & math
{'Cy', 'Dee'}
>>> cs | math
{'Ana', 'Ben', 'Cy', 'Dee', 'Eli', 'Fay'}
>>> cs - math
{'Ana', 'Ben'}
>>> math - cs
{'Eli', 'Fay'}
```



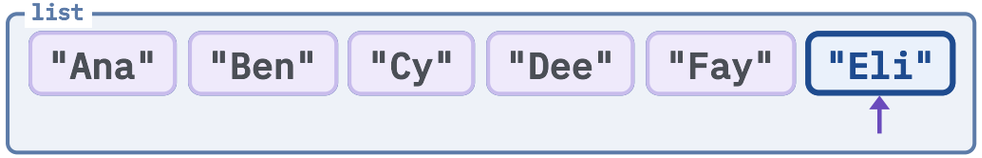
- `&` In both: the **intersection**
- `|` In either: the **union**
- `-` In the first, not the second: the **difference**

- Also as methods: `cs.intersection(math)`, `cs.union(math)`, `cs.difference(math)`

Is Eli here?

A LIST

```
"Eli" in signins
```

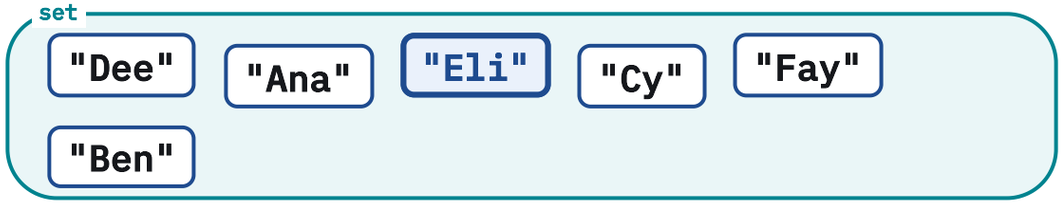


Checks one name at a time

6 checks to find Eli

A SET

```
"Eli" in here
```



Goes straight to where Eli would be

1 check

- A set uses the value itself to find its spot
  How? Take CS 201 to learn how data structures like sets are built
- 1,000,000 numbers: about 7 ms for a list, 0.00003 ms for a set
  Dictionary keys are fast in the same way

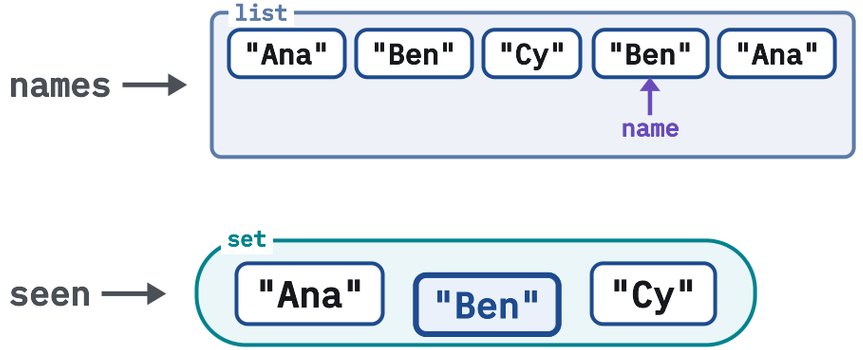
Who signed in twice first?

```
def first_repeat(names):
    seen = set()
    for name in names:
        if name in seen:
            return name
        seen.add(name)
    return None

print(first_repeat(["Ana", "Ben", "Cy", "Ben", "Ana"]))
```

OUTPUT

```
Ben
```



- `set()` starts an empty set
- Each name goes in `seen` once
- `in` on a set: one quick check
- Return as soon as a name repeats
  No repeats: `None`

Which would you use?

	List	Tuple	Dictionary	Set
Looks like	`[3, 7]`	`(3, 7)`	`{"Ana": 92}`	`{3, 7}`
Find an item by	Position	Position	Key	Only `in`
Can change?	Yes	No	Yes	Yes
Duplicates?	Yes	Yes	Not as keys	No

Your to-do list, most important first **List**

The different letters in a word **Set**

The height of each Vermont peak **Dictionary**

One student's name, email, and class year **Tuple**

The students who turned in Homework 4 **Set**

Part 2 of 3

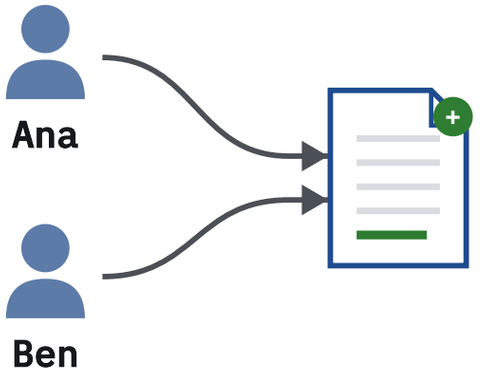
References

A variable holds a **reference** to an object, not the object itself

Two names can refer to the same object

Sharing a link vs. sending a copy

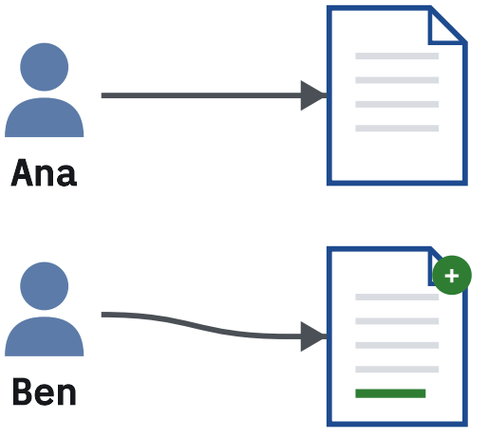
SHARE A LINK TO A GOOGLE DOC



Ben edits the doc

Ana sees his edit

EMAIL A COPY OF THE FILE



Ben edits his copy

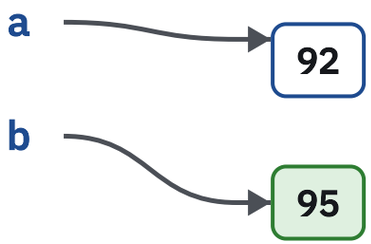
Ana's file doesn't change

- A Python variable is like the link
  It holds a **reference** to an object
- `b = a` shares the link: `b` is an **alias**, a second name for the object
  `b = a.copy()` sends a copy

Two names, one object

A NUMBER

```
a = 92
b = a
b = 95
print(a)
```



OUTPUT

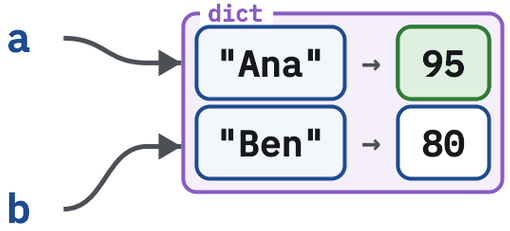
```
92
```

`b = 95` names a new number

`a` still names 92

A DICTIONARY

```
a = {"Ana": 92, "Ben": 80}
b = a
b["Ana"] = 95
print(a)
```



OUTPUT

```
{'Ana': 95, 'Ben': 80}
```

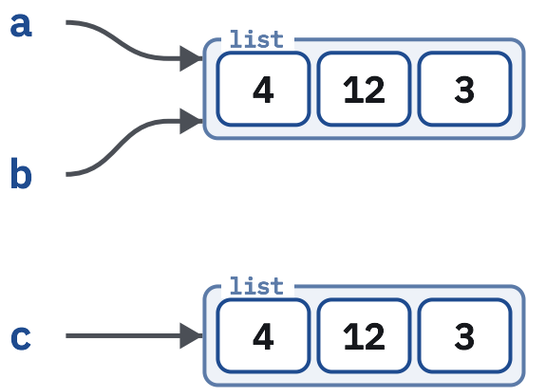
`b["Ana"] = 95` changes the dictionary

Every name for it sees the change

- Numbers, strings, and tuples are immutable, so sharing them is harmless

Same value or same object?

```
>>> a = [4, 12, 3]
>>> b = a
>>> c = [4, 12, 3]
>>> a == c
True
>>> a is c
False
>>> a is b
True
>>> id(a), id(b), id(c)
(4371254528, 4371254528, 4371256000)
```



- `==` asks: the same value?
- `is` asks: the same object?
- `id(x)` is a number that names an object
  Same `id` : same object

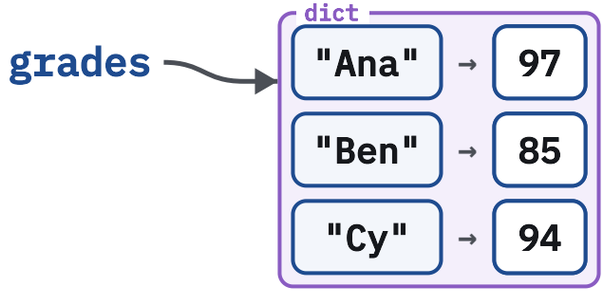
A function gets a reference

```
def curve(scores):
    for name in scores:
        scores[name] += 5

grades = {"Ana": 92, "Ben": 80, "Cy": 89}
curve(grades)
print(grades)
```

OUTPUT

```
{'Ana': 97, 'Ben': 85, 'Cy': 94}
```



- Inside the call, `scores` is a second name for `grades`
- No `return` needed: the caller sees the change
- To keep the original, pass a copy `curve(grades.copy())`

Part 3 of 3

Lists of lists and copies

A list can hold other lists, like the rows of a game board

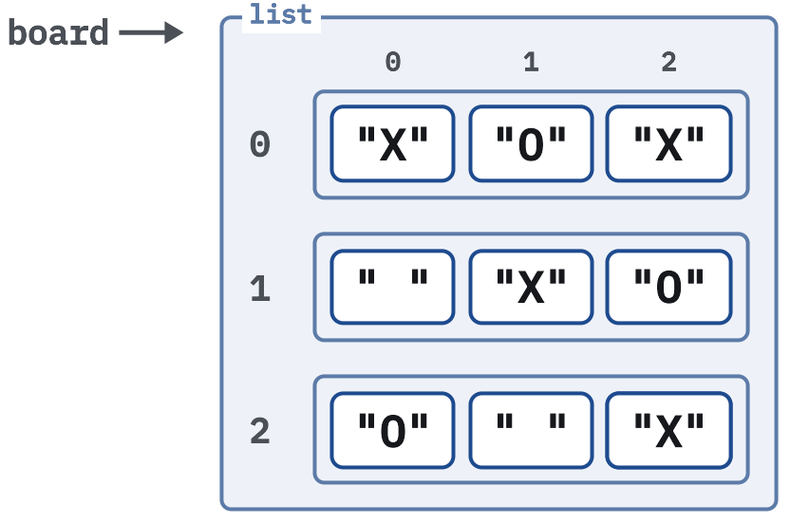
Copying one takes some care

A list of lists

```
board = [["X", "O", "X"],
         [" ", "X", "O"],
         ["O", " ", " "]]
print(board[1])
print(board[1][2])
board[2][2] = "X"
print(board[2])
```

OUTPUT

```
[' ', 'X', 'O']
O
['O', ' ', 'X']
```

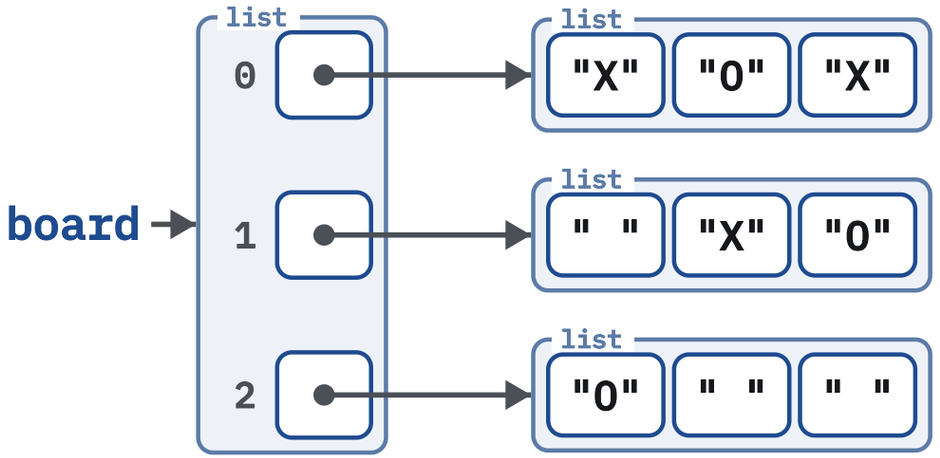


- Each item of `board` is a row: a list
- `board[1][2]`: row 1, then item 2 of that row
- A row can change like any list
  X wins!

Inside a list of lists

```
board = [["X", "O", "X"],
         [" ", "X", "O"],
         ["O", " ", " "]]
```

- The outer list holds three references
  one to each row
- Each row is a separate list
- To copy `board`, do we copy the references
  or the rows too?

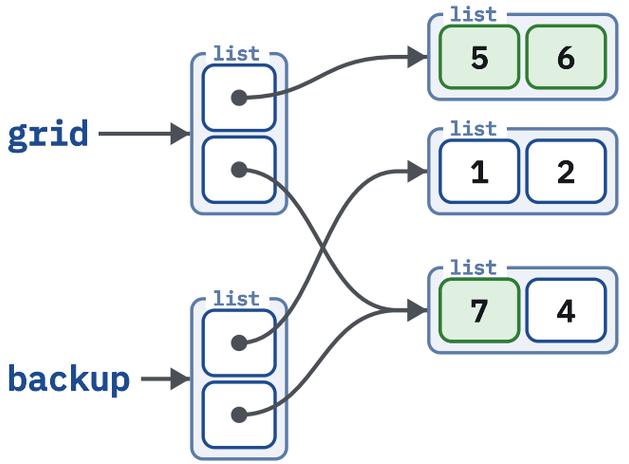


Copying a list of lists

```
grid = [[1, 2], [3, 4]]
backup = grid.copy()
grid[0] = [5, 6]
grid[1][0] = 7
print(grid)
print(backup)
```

OUTPUT

```
[[5, 6], [7, 4]]
[[1, 2], [7, 4]]
```



- `.copy()` makes a new outer list
  The rows inside are shared
  This is a **shallow copy**
- Replacing a row changes only `grid`
- A change inside a shared row
  shows up through both names

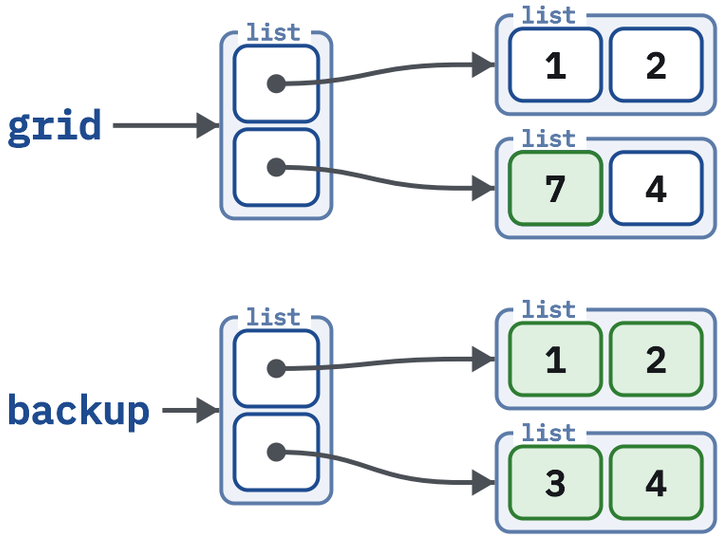
A deep copy

```
import copy
grid = [[1, 2], [3, 4]]
backup = copy.deepcopy(grid)
grid[1][0] = 7
print(backup)
```

OUTPUT

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

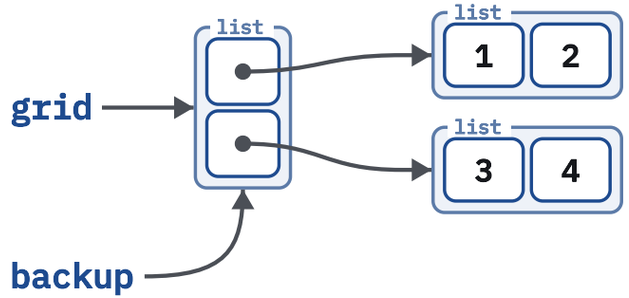
- `copy.deepcopy` copies the lists inside too all the way down: a **deep copy**
- Nothing is shared, so the 7 stays in `grid`
- Needs `import copy`



Alias, shallow copy, or deep copy?

ALIAS

```
backup = grid
```

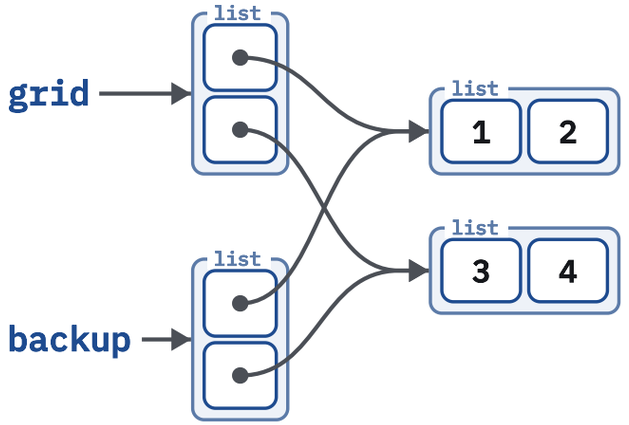


A second name for the same list

Every change shows up in both

SHALLOW COPY

```
backup = grid.copy()
```



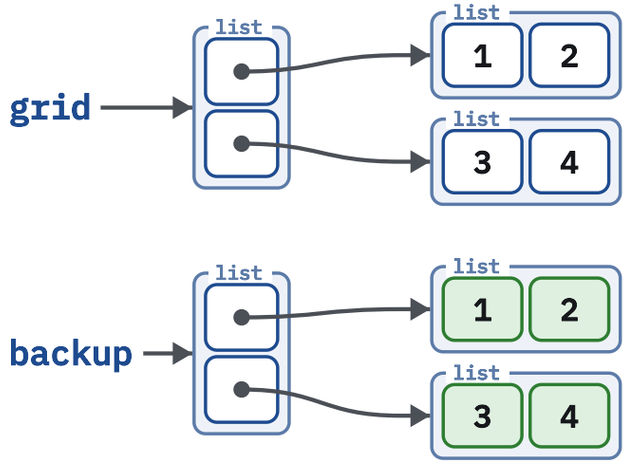
A new outer list

The lists inside are shared

Also `grid[:]` and `list(grid)`

DEEP COPY

```
backup = copy.deepcopy(grid)
```



New lists all the way down

Nothing is shared

- A list of numbers or strings: a shallow copy is all you need

  They're immutable, so sharing them is harmless

What does this print?

```
rows = [[5, 6], [7, 8]]
saved = rows.copy()
saved[0] = [0, 0]
saved[1][1] = 9
print(rows)
```

A `[[5, 6], [7, 8]]`

B `[[0, 0], [7, 9]]`

C `[[5, 6], [7, 9]]`

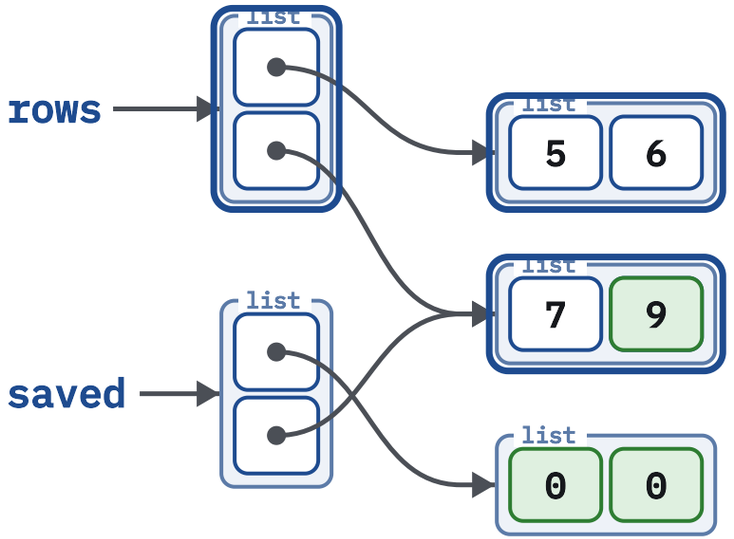
D `[[0, 0], [7, 8]]`

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

Or join directly here

What does this print?

```
rows = [[5, 6], [7, 8]]
saved = rows.copy()
saved[0] = [0, 0]
saved[1][1] = 9
print(rows)
```



A `[[5, 6], [7, 8]]`

B `[[0, 0], [7, 9]]`

C `[[5, 6], [7, 9]]`

D `[[0, 0], [7, 8]]`

Answer: C

`saved[0] = [0, 0]` changes `saved` only
`saved[1]` is `rows[1]`, so `rows` gets the 9

Why not the others?

A: treats `.copy()` as a deep copy
B: treats `saved` as an alias
D: has both backwards

Activity

1. Download day13.zip.
2. Drag `day13.zip` to your `class` folder and extract it there.
   Mac — double-click it. **Windows** — right-click it and choose "Extract All...".
   Once you see the `day13` folder, delete `day13.zip`.
3. Launch Thonny. From the File menu, open `cs145/class/day13/day13_sets_refs.py`.
   The code from today's slides is in `day13_demos.py`.
4. Write `has_duplicates`, `in_both`, and `copy_grid`.
5. If you have time, write `missing_letters`.

Patterns for the activity

SETS

Make a set from a list

```
unique = set(nums)
```

Count different values

```
len(set(nums))
```

In both

```
set(first) & set(second)
```

In the first only

```
set(first) - set(second)
```

Sorted list from a set

```
sorted(unique)
```

Letters in a string

```
set(word.lower())
```

LISTS OF LISTS

Copy a list

```
new_row = row.copy()
```

Build a new list

```
result = []
for row in grid:
    result.append(...)
return result
```

To do

- Review this slide deck and section 3 of Notes 11
- Homework 4 is due Thursday 10/29

Next time

Lecture 14 — files and file formats