

Lecture 9

Other Collections and Nested Data

Lecture 9 - Collections

- We already know lists
- Now we ask: which data structure best represents our data?
 - **tuple**: fixed ordered data
 - **set**: unique unordered elements
 - **dict**: key-value mapping
 - **nested collections**: tables and records

Different structures are useful for different problems.

Lecture 9 - Collections

- Choosing a structure
 - Need **order** and **mutability**? use a **list**
 - Need a **fixed group** (immutable sequence) of values? use a **tuple**
 - Need **uniqueness** and fast membership? use a **set**
 - Need lookup by **key**? use a **dictionary**.

```
grades = [15, 18, 12]
point = (3, 4)
unique_ids = {"A12", "B07"}
prices = {"pen": 1.20, "book": 12.50}
```

Tuples

Fixed ordered sequences of values

Lecture 9 - Tuples

- What is a tuple?
 - An ordered collection of values
 - Similar to a list, but **immutable**
 - Useful for fixed records or coordinates
 - created using ()
 - Access by index works as with lists.
- NOTE: once created, their elements cannot be added, removed, or replaced. However, if an element is mutable, it can change!

```
point = (3, 4)

point[0]    # 3
point[1]    # 4

# point[0] = 5  # error

sequences = ([1, 2], 10)
sequences[0].append(3)  # allowed!
```

Tuple = ordered but not modifiable.

Lecture 9 - Tuples

- Tuple unpacking
 - A tuple can be **unpacked** into variables
 - The number of variables must match the number of elements
 - This is common with `enumerate()`
 - `enumerate()` returns a tuple (index, value)

```
point = (3, 4)
x, y = point

for i, value in enumerate([10, 20]):
    print(i, value)
```

We already used unpacking with `enumerate()`.

Sets

Unique elements without positions

Lecture 9 - Sets

- What is a set?
 - A **mutable** collection of **unique elements**
 - Duplicates are automatically removed
 - Elements are not stored in a meaningful order
 - You cannot access elements by position.

```
ids = {"A12", "B07", "A12"}  
print(ids)  
# {"A12", "B07"}  
  
# ids[0]  # error
```

Set = uniqueness, not order.

Lecture 9 - Sets

- Creating sets
 - Use { } with elements inside
 - Use set() to create an empty set
 - Use set(sequence) to remove duplicates
→ obtain a set containing the unique elements of the sequence

```
colors = {"red", "green", "blue"}  
empty = set()
```

```
names = ["Ana", "Bob", "Ana"]  
unique_names = set(names)  
#{"Ana", "Bob"}
```

NOTE: {} creates an empty dictionary, not an empty set.

Lecture 9 - Sets

- What can a set contain?
 - A set, because of his properties, can contain only immutable objects (or better hashable...)
 - int, float, bool
 - str
 - tuple, only if all the elements are immutable
 - frozenset (an immutable set)

```
colors = {"red", "green", "blue"}  
empty = set()
```

```
names = ["Ana", "Bob", "Ana"]  
unique_names = set(names)  
#{"Ana", "Bob"}
```

NOTE: {} creates an empty dictionary, not an empty set.

<https://docs.python.org/3/builtins/stdtypes.html#set-types-set-frozenset>

Lecture 9 - Sets

- Membership in sets
 - Use `in` and `not in`
 - The result is Boolean
 - Sets are excellent for membership checks
 - We can iterate on set elements using
`for <element> in <set_name>:`

```
allowed_codes = {"A", "B", "C"}
code = input("Code: ").upper()

if code in allowed_codes:
    print("Accepted")
else:
    print("Rejected")
```

Useful for input validation.

Lecture 9 - Sets

- Adding and removing elements
 - `.add(value)` adds an element
 - `.discard(value)` removes if present
 - `.remove(value)` raises an error if absent
 - `.clear()` removes all elements.

```
codes = set()
codes.add("A12")
codes.add("B07")
codes.add("A12")

codes.discard("B07")
```

Adding a duplicate has no effect.

Lecture 9 - Sets

- Checking Set Inclusion

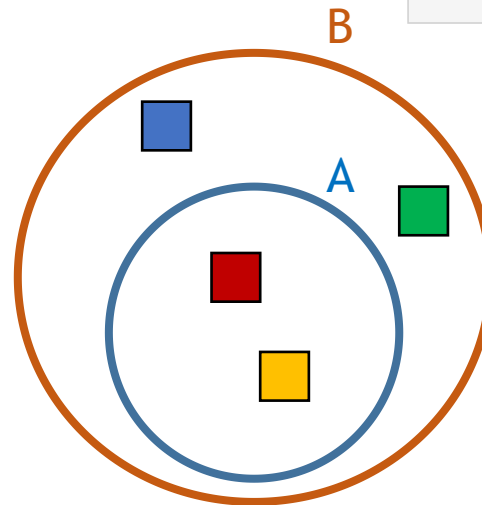
- `.issubset()` checks whether all elements of one set (the one who calls) are contained in another set (given as argument)
- The opposite is `.issuperset()`
- They return a boolean

```
A = {1, 2}
```

```
B = {1, 2, 3, 4}
```

```
A.issubset(B)      # True
```

```
B.issubset(A)      # False
```



Lecture 9 - Sets

- Set Comparison
 - Different comparison operators can be used with sets
 - They can be combined with and/or/not

Operator	Meaning
<code>A <= B</code>	A is a subset of B
<code>A < B</code>	A is a proper subset of B
<code>A >= B</code>	A is a superset of B
<code>A > B</code>	A is a proper superset of B
<code>A == B</code>	Same elements (equal)
<code>A != B</code>	Different elements (unequal)

```
A = {1, 2}
B = {1, 2, 3, 4}
```

```
A <= B    # True
A <  B    # True
A >= B    # False
A >  B    # False
A == B    # False
A != B    # True
```

Equivalent to `.issubset()`
and `.issuperset()`

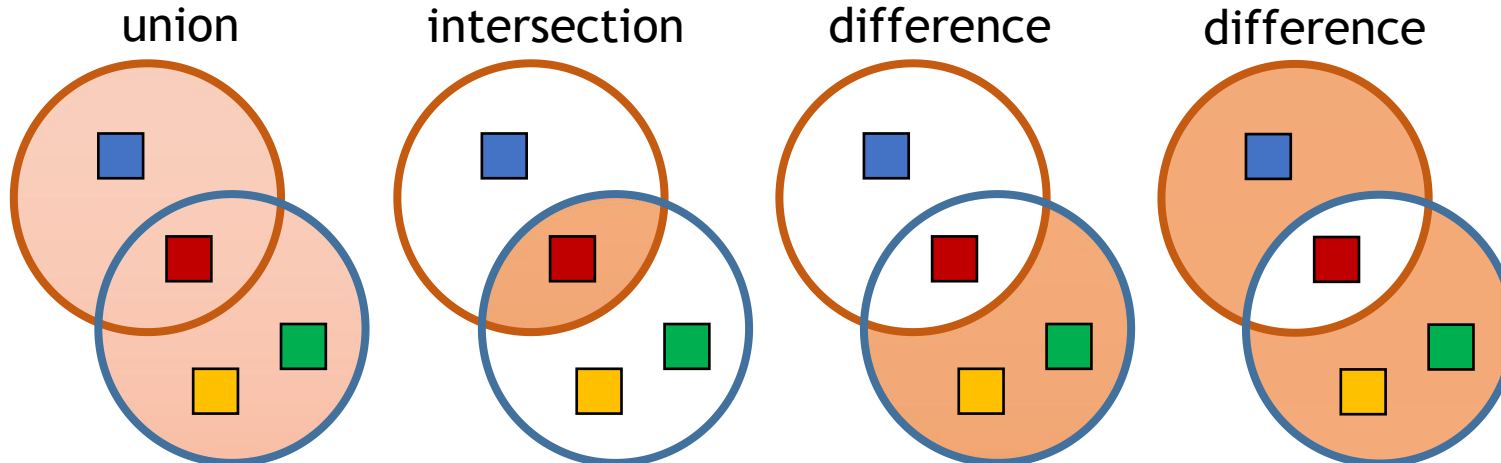
Lecture 9 - Sets

- Set operations
 - `.union()`: elements in either set
 - `.intersection()`: elements in both sets
 - `.difference()`: elements in the first but not in the second
 - `.symmetric_difference()`: elements in the first or second but not in both

```
a = {"red", "yellow", "green"}  
b = {"red", "blue"}
```

```
a.union(b)          # r, y, b, g  
a.intersection(b)   # r  
a.difference(b)     # y, g  
a.symmetric_difference(b) # y, g, b
```

These mirror mathematical set operations.



Lecture 9 - Sets

- Set operations
 - Sets support also different operators, equivalent to the corresponding methods
 - as usual, they can be combined in more complex expressions
 - both operators and methods return a **new set**

```
a = {"red", "yellow", "green"}  
b = {"red", "blue"}
```

```
a | b      # r, y, b, g  
a & b      # r  
a - b      # y, g  
a ^ b      # y, g, b
```

Operator	Method	Meaning
A B	A.union(B)	Elements in A or B
A & B	A.intersection(B)	Elements in A and B
A - B	A.difference(B)	Elements in A and not in B
A ^ B	A.symmetric_difference(B)	elements that are in A or B, but not both

Lecture 9 - Sets

- Why using sets (instead of lists always)
 - If we need unique elements
 - If we need to do set operation
 - **Efficiency:** fast membership check (up to 10×)

if item not in item_list
implies a scan of all the list

```
# not efficient:  
if item not in item_list:  
    item_list.append(item)  
  
# same result, but more efficient  
item_set.add(item)
```

Dictionaries

Lookup by key

Lecture 9 - Dictionaries

- What is a dictionary?
 - A collection of **key-value** pairs
 - Each key is **unique**
 - Values are accessed through keys, not positions, using []
 - Useful as a lookup table
 - Dictionaries are **mutable** → we can change values and add/remove couples
 - Like lists, dictionaries preserve **insertion order**.

```
prices = {  
    "pen": 1.20,  
    "book": 12.50,  
    "bag": 25.00  
}  
  
prices["book"]    # 12.50
```

Dictionary = key → value.

Lecture 9 - Dictionaries

- Keys are in general
 - String
 - names
 - fields
 - ID codes...
 - Numbers
 - numeric IDs
- Values can be of any type

Use dictionaries when a meaningful key identifies the value.

```
prices = {  
    "pen": 1.20,  
    "book": 12.50,  
    "bag": 25.00  
}  
  
grades = {  
    172834: 14,  
    173422: 18  
}  
  
bag_price = prices["bag"]  
my_grade = grades[173422]
```

Lecture 9 - Dictionaries

- Creating and updating dictionaries

- to create a dict, use {key: value, ...}
- {} or dict() to initialize an empty dict
- Assignment of a value with =
 - Assigning to an **existing key** updates the value
 - Assigning to a **new key** adds a **new pair**
- To copy a dictionary:

```
copyOfDict = dict(old_dict)
```

- not copyOfDict = old_dict that is a reference to the same object

```
stock = {"pen": 30, "book": 12}

stock["pen"] = 25      # update
stock["notebook"] = 40 # add

print(stock)
# {"pen": 30, "book": 12,
#  "notebook": 40}
```

Keys must be unique.

Lecture 9 - Dictionaries

- Avoiding missing-key errors
 - Accessing a missing key with [] raises `KeyError`
 - Use `in` to check whether a key exists
 - Or use `.get(key)` to provide a default value.
 - if key exists, it returns the associated value
 - if key do not exist, it returns the default value

```
prices = {"pen": 1.20}

if "book" in prices:
    print(prices["book"])
else:
    print("Unknown product")

prices.get("book", 0)
```

**Validate the key before using it,
or use `.get()` and check the output**

Lecture 9 - Dictionaries

- Iterating over dictionaries
 - `for key in dict` iterates over keys
 - `dict.items()` gives key-value pairs
→ we can iterate on pairs with
`for key, val in dict.items()`
 - `dict.values()` gives only values
→ we can efficiently iterate on values with
`for val in dict.values()`

```
prices = {"pen": 1.20, "book": 12.50}

for product in prices:
    print(product, prices[product])

for product, price in prices.items():
    print(product, price)
```

`items()` is often the most readable option.

Lecture 9 - Dictionaries

- Removing items with `.pop()`
 - Similar to lists, but using keys instead of indexes
 - `dict.pop(key)` **removes** the specified key and **returns its value**
 - If key do not exist it raises a `KeyError`
 - To avoid errors, use a **default return value** as a second argument, and check the result.

```
student = {  
    "name": "Alice",  
    "grade": 18,  
    "year": 2  
}  
  
grade = student.pop("grade")  
  
print(grade)      # 18  
print(student)    # {'name': 'Alice', 'year': 2}  
  
student.pop("age") # KeyError  
age = student.pop("age", None)
```

Lecture 9 - Dictionaries

- Getting dict length with `len()`
 - As for lists, returns the number of key-values pairs

```
student = {  
    "name": "Alice",  
    "grade": 18,  
    "year": 2  
}  
  
len(student)    # 3
```

Lecture 9 - Dictionaries

- Iterating over a dict in sorted **key** order
 - A dictionary preserves insertion order, but it is not automatically sorted
 - the function `sorted()` produces a **new list with sorted keys** (as default behaviour)
 - iterate over sorted keys with `for k in sorted(dictionary):`

```
grades = {  
    "Bob": 14,  
    "Alice": 18,  
    "Carol": 16  
}  
  
for name in sorted(grades):  
    print(name, grades[name])
```

output:

```
Alice 18  
Bob 14  
Carol 16
```

Lecture 9 - Sorted Lists and Dictionaries

- Short digression about `sorted()`
 - `sorted()` normally compares the elements of a sequence directly
 - The optional `key` parameter allows to specify which value should be used for the comparison:
 - For each element, Python **evaluates** the function given to `key` (using the element as an argument) and sorts **according to the returned value**
 - Example: *sorting a list of strings by their length*

```
words = ["pear", "banana", "fig"]  
  
sorted(words, key=len)  
# ['fig', 'pear', 'banana']
```

```
sorted(iterable, /, *, key=None, reverse=False)
```

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

<https://docs.python.org/3/builtins/functions.html#sorted>

Lecture 9 - Sorted Lists and Dictionaries

- Short digression about `sorted()`
 - `sorted()` normally compares the elements of a sequence directly
 - The optional `key` parameter allows to specify which value should be used for the comparison:

- For each element, Python **evaluates** the function given to `key` (using the element as an argument) and sorts **according to the returned value**

- Example: *sorting a list of strings by their length*

```
words = ["pear", "banana", "fig"]  
  
sorted(words, key=len)  
# ['fig', 'pear', 'banana']
```

`sorted(iterable, /, *, key=None, reverse=False)`

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Note: do not use (): we are not *calling* the function, the sorted procedure is doing it!

Here, we pass the name of the function as an object

<https://docs.python.org/3/builtins/functions.html#sorted>

Lecture 9 - Dictionaries

- Iterating by sorted dictionary **values**
 - We can exploit sorted with a `key=...` argument
 - we need a **function** or a **method** that, given the key, returns the associated value...

Which is this method?

Lecture 9 - Dictionaries

- Iterating by sorted dictionary **values**
 - We can exploit sorted with a `key=...` argument
 - To iterate according to the associated values, use the dictionary method `.get` as the sorting key.

```
grades = {  
    "Bob": 14,  
    "Alice": 18,  
    "Carol": 16  
}  
  
for name in sorted(grades, key=grades.get):  
    print(name, grades[name])
```

output:

Alice 18
Carol 16
Bob 14

Lecture 9 - Dictionaries

- Examples of dictionary usage in different styles:

```
# Associative array / lookup table
prices = {
    "apple": 1.20,
    "banana": 0.80,
    "orange": 1.50
}

prices["banana"]    # 0.80
```

```
# Mapping codes to meanings
status = {
    "P": "Pending",
    "A": "Approved",
    "R": "Rejected"
}
```

```
# Record describing one object
student = {
    "id": 1042,
    "name": "Alice",
    "grade": 18
}

# then, we may have list of students
```

```
# Counting occurrences
sales = {
    "Laptop": 12,
    "Mouse": 47,
    "Keyboard": 25
}
```

Nested Lists as Tables

Rows, columns and nested loops

Lecture 9 - Nested Data

- List of lists
 - A table can be represented as a list of rows.
 - Each row is itself a list.
 - Access uses two indexes: [row][column].

	col 0	col 1	col 2
row 0	15	18	12
row 1	17	14	19
row 2	11	16	13

```
grades = [  
    [15, 18, 12],  
    [17, 14, 19],  
    [11, 16, 13]  
]
```

```
grades[1]      # [17, 14, 19]  
grades[1][2]   # 19
```

First select the row, then the column.

Lecture 9 - Nested Data

- Iterating over a table
 - To iterate on nested lists we need **nested loops**
 - The outer loop scans rows
 - The inner loop scans values inside each row
 - This is a natural use of nested loops.

```
for row in grades:  
    for value in row:  
        print(value)  
  
# one row at a time  
for row in grades:  
    print(row)
```

Nested data often requires nested loops.

Lecture 9 - Nested Data

- Row and column summaries
 - A row average is easy: use one row at a time
 - A column summary requires fixed column indexes
 - Be careful with row and column meanings.

```
for row in grades:  
    avg = sum(row) / len(row)  
    print(avg)  
  
# first column only  
total = 0  
for row in grades:  
    total = total + row[0]
```

Tables force us to reason about dimensions.

Nested Collections

Combining lists and dictionaries

Lecture 9 - Nested Data

- List of dictionaries
 - Useful for records with the same fields
 - Each dictionary represents one item
 - The list stores many records.

```
students = [  
    {"name": "Ana", "id": 17346, "grade": 17},  
    {"name": "Bob", "id": 17553, "grade": 14}  
]  
  
students[0]["name"] # Ana
```

Very common in data-oriented programs.

Lecture 9 - Nested Data

- Dictionary of lists
 - Useful when each key is associated with several values
 - For example: grades by course or sales by month
 - The key selects the group; the list stores the values.

```
grades_by_course = {  
    "Math": [15, 18, 12],  
    "Economics": [14, 16]  
}  
  
values = grades_by_course["Math"]  
avg = sum(values) / len(values)
```

A dictionary can organize multiple lists.

Lecture 9 - Nested Data

- Dictionary of dictionaries
 - Useful for lookup tables with structured values
 - The first key selects the record
 - The second key selects the field.

```
products = {  
    "P001": {"name": "Pen", "price": 1.20},  
    "P002": {"name": "Book", "price": 12.50}  
}  
  
products["P002"]["price"] # 12.50
```

Powerful, but keep it readable.

Lecture 9 - Nested Data

- Why not use more than two levels?

```
class_results = {
    "Alice": {
        "Programming": [18, 17, 20],
        "Mathematics": [16, 18]
    },
    "Bob": {
        "Programming": [15, 18],
        "Mathematics": [14, 16, 17]
    }
}

class_results["Alice"]
class_results["Alice"]["Programming"]
class_results["Alice"]["Programming"][0]    # 18
```

Lecture 9 - Nested Data

- Choosing the right representation
 - Table with rows and columns → list of lists
 - Unique values → set
 - Lookup by identifier → dictionary
 - Many records with fields → list of dictionaries
 - Groups of values by key → dictionary of lists

The structure should match the question you need to answer.

Lecture 9 - Nested Data

Poor representation

```
names = ["Ana", "Bob"]  
grades = [17, 14]  
  
# Which grade belongs to Ana?  
# Must keep two lists synchronized
```

Better representation

```
students = [  
    {"name": "Ana", "grade": 17},  
    {"name": "Bob", "grade": 14}  
]  
  
# Each record keeps its fields  
# together
```

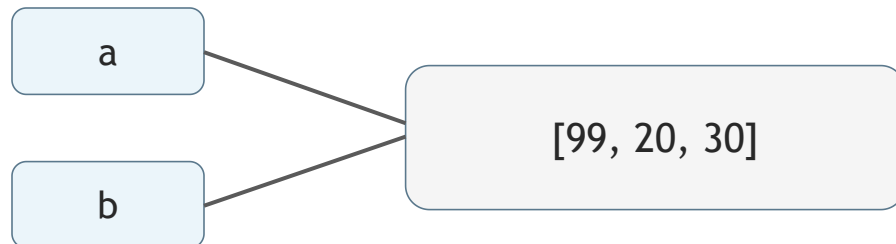
Readable data structures reduce errors.

Nested Mutables and Deep Copy

Why shallow copy is not enough for nested collections

Recall: Aliasing with a Mutable Object

Assigning a list to another variable does not copy the list
Both names refer to the same object in memory
Mutating the list through one name is visible through the other.



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

```
b = a
```

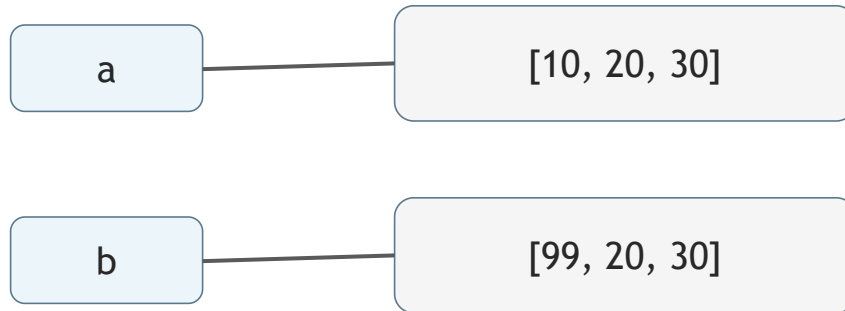
```
b[0] = 99
```

```
print(a)    # [99, 20, 30]
```

Two variables, one list object

Shallow Copy Solves the Flat Case

A *shallow copy* creates a new outer container
For a flat list of immutable values, this is
usually enough
Changing one list does not change the other.



```
a = [10, 20, 30]
b = a.copy()      # or list(a), a[:]
```

```
b[0] = 99
print(a)    # [10, 20, 30]
print(b)    # [99, 20, 30]
```

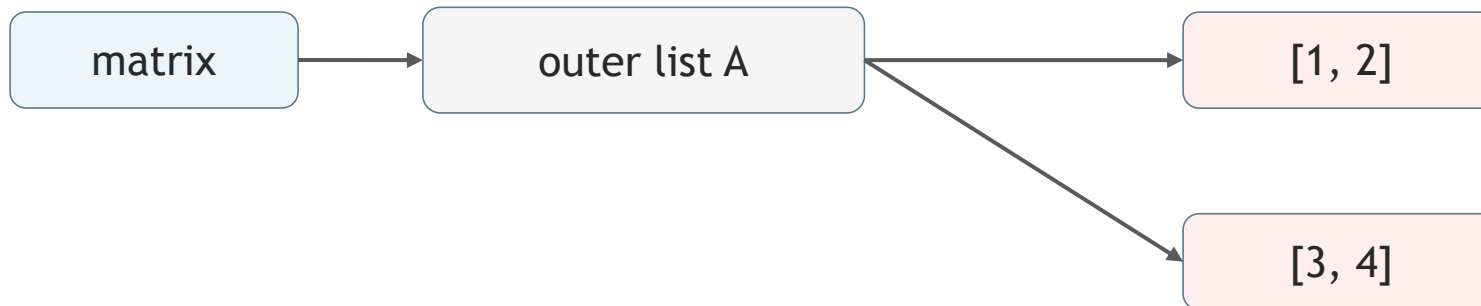
New outer list → independent mutation

Nested Mutables: The Problem Returns

Common error with nested mutables (part 3 of 3634943298478...)



A nested list contains other mutable objects
A shallow copy creates a new outer list only
The inner lists are still shared references.



```
matrix = [[1, 2], [3, 4]]
copy_matrix = matrix.copy()

copy_matrix.append([0, 0])

copy_matrix[0][0] = 99

print(matrix)
# [[99, 2], [3, 4]]
print(copy_matrix)
# [[99, 2], [3, 4], [0, 0]]
```

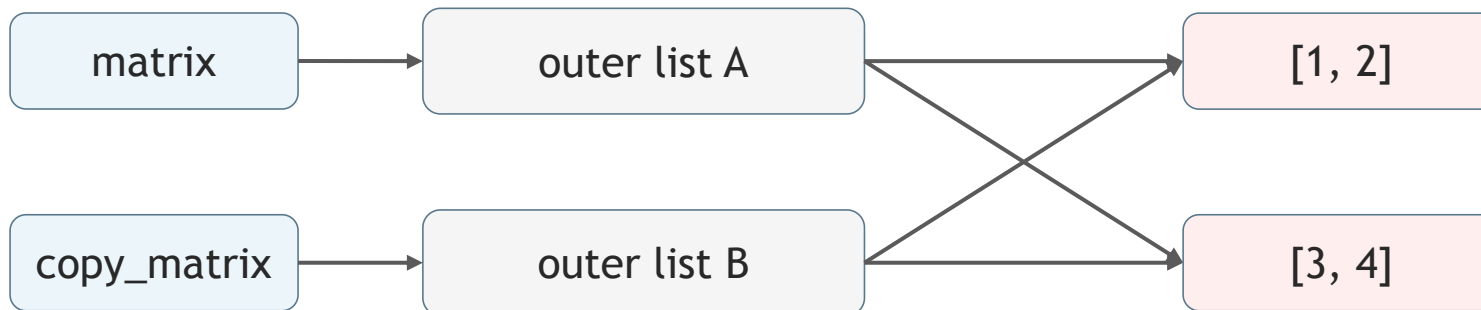
Only the first level
was copied.

Nested Mutables: The Problem Returns

Common error with nested mutables (part 3 of 3634943298478...)



A nested list contains other mutable objects
A shallow copy creates a new outer list only
The inner lists are still shared references.



```
matrix = [[1, 2], [3, 4]]
copy_matrix = matrix.copy()

copy_matrix.append([0, 0])

copy_matrix[0][0] = 99

print(matrix)
# [[99, 2], [3, 4]]
print(copy_matrix)
# [[99, 2], [3, 4], [0, 0]]
```

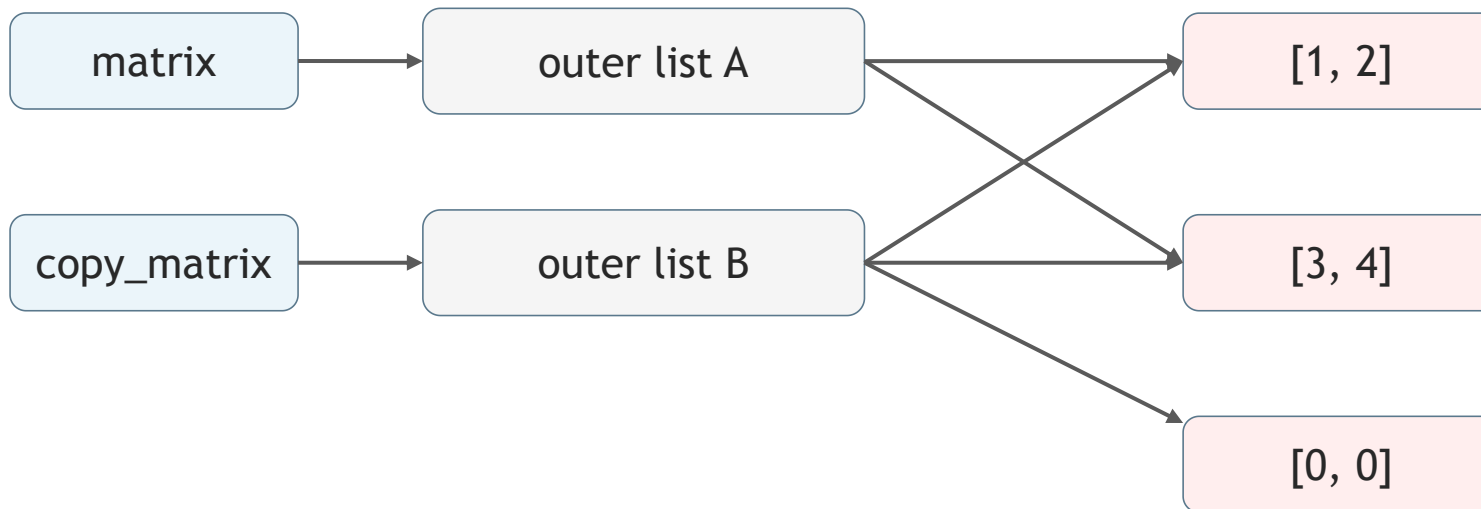
Only the first level
was copied.

Nested Mutables: The Problem Returns

Common error with nested mutables (part 3 of 3634943298478...)



A nested list contains other mutable objects
A shallow copy creates a new outer list only
The inner lists are still shared references.



```
matrix = [[1, 2], [3, 4]]
copy_matrix = matrix.copy()

copy_matrix.append([0, 0])

copy_matrix[0][0] = 99

print(matrix)
# [[99, 2], [3, 4]]
print(copy_matrix)
# [[99, 2], [3, 4], [0, 0]]
```

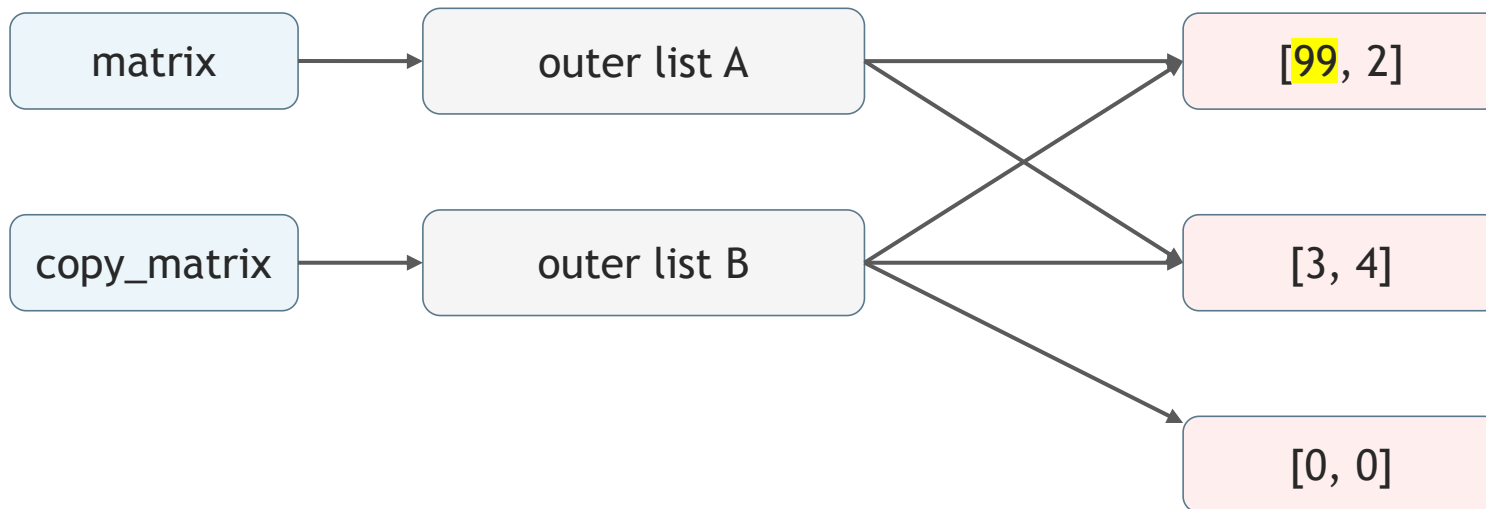
Only the first level
was copied.

Nested Mutables: The Problem Returns

Common error with nested mutables (part 3 of 3634943298478...)



A nested list contains other mutable objects
A shallow copy creates a new outer list only
The inner lists are still shared references.



```
matrix = [[1, 2], [3, 4]]
copy_matrix = matrix.copy()

copy_matrix.append([0, 0])

copy_matrix[0][0] = 99

print(matrix)
# [[99, 2], [3, 4]]
print(copy_matrix)
# [[99, 2], [3, 4], [0, 0]]
```

Only the first level
was copied.

Same Issue with Nested Dictionaries

Common error with nested mutables (part 3 of 3634943298478...)



The same idea applies to dictionaries
Copying a dictionary copies the outer
dictionary only
Nested lists or dictionaries may still be shared.

The value associated with "grades" is a list.

```
student = {  
    "name": "Alice",  
    "grades": [18, 17]  
}  
backup = dict(student)      # shallow copy  
  
backup["grades"].append(20)  
print(student["grades"])  
# [18, 17, 20]
```

Both dictionaries still refer to the same grade list.

Solution 1: Copy the Inner Level Manually

- If the structure is simple and known, copy the nested objects explicitly
- Useful when you know exactly which level must be independent
- Keeps the solution transparent for small structures.

```
# Same idea with a comprehension  
copy_matrix = [row.copy() for row in matrix]
```

```
matrix = [[1, 2], [3, 4]]  
copy_matrix = []  
  
for row in matrix:  
    copy_matrix.append(row.copy())  
  
copy_matrix[0][0] = 99  
print(matrix)  
# [[1, 2], [3, 4]]
```

New outer list + new inner lists

Solution 2: Use `copy.deepcopy()`

- For more complex nested structures, use the `copy` module
- `deepcopy()` recursively copies nested objects.
- The copy is independent at all levels that can be copied.

```
import copy

records = [
    {"name": "Alice", "grades": [18, 17]},
    {"name": "Bob", "grades": [14, 16]}
]
backup = copy.deepcopy(records)

backup[0]["grades"].append(20)
print(records[0]["grades"])
# [18, 17]
```

Use this when the nested structure is not trivial.

`deepcopy` is safer, but it can be heavier.

Rule of Thumb

- **Aliasing:** two names refer to the same object.
- **Shallow copy:** new outer container, shared nested objects.
- **Deep copy:** recursively copied nested objects.
- For nested mutable data, always ask: which levels must be independent?

```
b = a                # alias  
b = a.copy()         # shallow copy  
b = copy.deepcopy(a) # deep copy
```

Choose the copy strategy based on the structure you need.