

Lecture 7

Lists

Lecture 7 - Lists

- After loops, we need containers for many values.
 - Create and inspect lists
 - Access and **modify** elements
 - Iterate over lists with for
 - Build lists dynamically
 - Understand references, aliases and copies

A list is the first mutable collection we study.

Lecture 7 - Lists

- Why do we need lists?
 - Without lists, many related values require many variables
 - We may not know how many values and variables to declare
 - A list stores several values under one name
 - This makes loops useful for real data processing.

```
grade1 = 15  
grade2 = 18  
grade3 = 12
```

```
# better  
grades = [15, 18, 12]
```

From repeated variables to a collection.

Lecture 7 - Lists

- What is a list?
 - built-in type
 - An **ordered** collection of elements (sequence)
 - Elements can be accessed by position
 - Lists are **dynamic**: their size can change.
 - Lists are **mutable**: their elements **can be modified** (!= strings).

```
prices = [12.5, 8.0, 19.9]

prices[0]      # 12.5
prices[1]      # 8.0

prices[1] = 13.6 # modified!

print(prices)
# [12.5, 13.6, 19.9]
```

Position is called index.

<https://docs.python.org/3/builtins/stdtypes.html#typeseq-list>

Lecture 7 - Lists

List as an indexed sequence

```
scores = [10, 9, 7, 4, 5]
```

index	0	1	2	3	4
value	10	9	7	4	5
neg. index	-5	-4	-3	-2	-1

```
scores[0]    # first element  
scores[-1]   # last element
```

Indexing works like with strings,
but list elements can change.

Lecture 7 - Lists

- Creating lists
 - Use square brackets [] to create a list
 - Elements are separated by commas
 - An empty list contains no elements
 - Mixed types are allowed, but homogeneous lists are usually clearer.

```
numbers = [10, 20, 30]
names = ["Ana", "João"]
empty = []
mixed = [10, "text", True]
```

Use mixed lists only when there is a good reason.

Lecture 7 - Lists

- List vs. string
 - Both are **ordered** sequences
 - Both use **indexes** and **slicing**
 - A string contains characters and is **immutable**
 - A list can contain values of different types and is **mutable**.

Same indexing idea, different mutability.

```
text = "elephant"
text[0] = "b"    # error
text[:5] # eleph

animals = ["cat", "dog", "fish", "rabbit"]
animals[0] = "bird"
animals[1:3] # ['dog', 'fish']
```

Lecture 7 - Lists

- List vs. string
 - Both are **ordered** sequences
 - Both use **indexes** and **slicing**
 - A string contains characters and is **immutable**
 - A list can contain values of different types and is **mutable**.

Same indexing idea, different mutability.

Being mutable, we can modify the original list, for example changing a value at a certain position.

List length may be modified too, by adding/removing items

```
text = "elephant"
text[0] = "b"    # error
text[:5] # eleph

animals = ["cat", "dog", "fish", "rabbit"]
animals[0] = "bird"
animals[1:3] # ['dog', 'fish']
```

References and Copies

Why lists behave differently from numbers and strings

Lecture 7 - Lists

- Variables and list objects
 - A variable **refers** to a list object in memory.
 - The list object contains the actual elements.
 - More than one variable can refer to the same list.

```
scores = [10, 9, 7]
```

scores refers to a list object

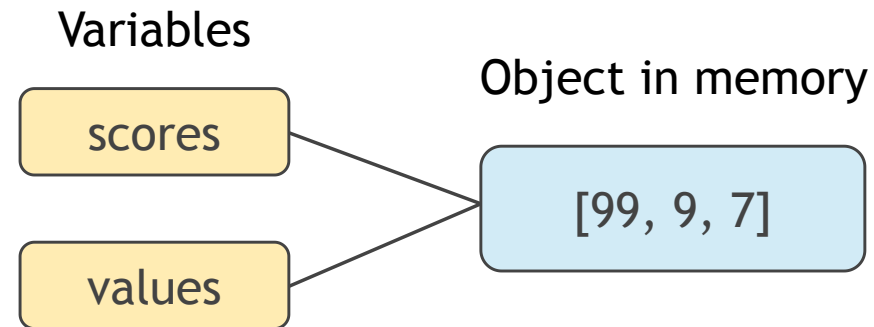
Important for understanding aliasing.

Lecture 7 - Lists

Aliasing

```
scores = [10, 9, 7]
values = scores

values[0] = 99
print(scores)    # [99, 9, 7]
```



Assignment copies the reference, not the list content.

Lecture 7 - Lists

Common error with nested mutables (part 1 of 3634943298478...)



- Meaning of an assignment statement:
 - Assigning a variable to another variable does not copy the object: both variables **refer to the same object** in memory (*aliasing*).
- List (mutable) vs. string (immutable):
 - A **mutable** object can be modified → the changes are propagated to **all** its references (assignment not safe)
 - An **immutable** object cannot be modified: an operation that changes its value creates a **new object** and reassigns the variable to it. (assignment safe)

Lecture 7 - Lists

Common error with nested mutables (part 1 of 3634943298478...)



- Meaning of an assignment statement:
 - Assigning a variable to another variable does not copy the object: both variables **refer to the same object** in memory (**aliasing**).
- List (mutable) vs. string (immutable):
 - A **mutable** object can be modified → the changes are propagated to **all** its references (assignment not safe)
 - An **immutable** object cannot be modified: an operation that changes its value creates a **new object** and reassigns the variable to it. (assignment safe)

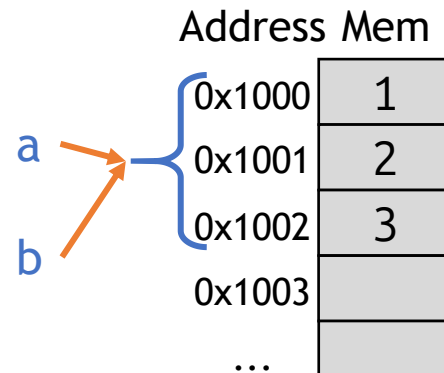
lists

```
a = [1, 2, 3]
```

```
b = a
```

```
b[0] = 10
```

```
print(a)    # [10, 2, 3]
```



Lecture 7 - Lists



Common error with mutables (part 1 of 3634943298478...)

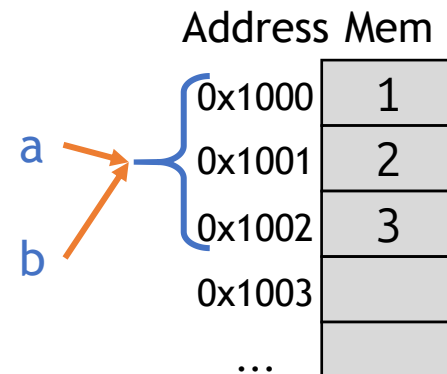
- Meaning of an assignment statement:
 - Assigning a variable to another variable does not copy the object: both variables **refer to the same object** in memory (**aliasing**).
- List (mutable) vs. string (immutable):
 - A **mutable** object can be modified → the changes are propagated to **all** its references (assignment not safe)
 - An **immutable** object cannot be modified: an operation that changes its value creates a **new object** and reassigns the variable to it. (assignment safe)

lists

```
a = [1, 2, 3]
b = a

b[0] = 10

print(a)    # [10, 2, 3]
```

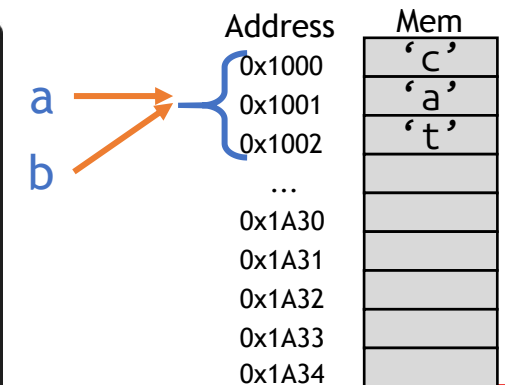


strings

```
a = "cat"
b = a

b = b + "s" # ref to new string

print(a)    # cat
print(b)    # cats
```



Lecture 7 - Lists



Common error with mutables (part 1 of 3634943298478...)

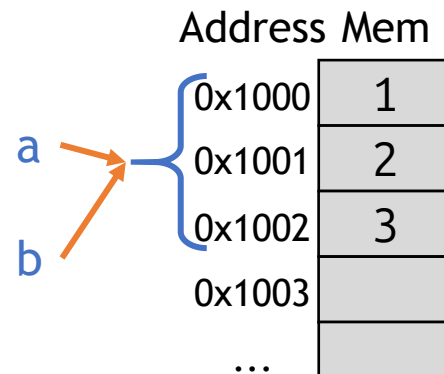
- Meaning of an assignment statement:
 - Assigning a variable to another variable does not copy the object: both variables **refer to the same object** in memory (**aliasing**).
- List (mutable) vs. string (immutable):
 - A **mutable** object can be modified → the changes are propagated to **all** its references (assignment not safe)
 - An **immutable** object cannot be modified: an operation that changes its value creates a **new object** and reassigns the variable to it. (assignment safe)

lists

```
a = [1, 2, 3]
b = a

b[0] = 10

print(a)    # [10, 2, 3]
```

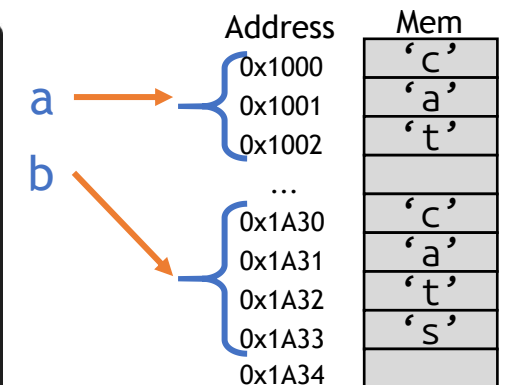


strings

```
a = "cat"
b = a

b = b + "s" # ref to new string

print(a)    # cat
print(b)    # cats
```



Lecture 7 - Lists

- Copying a list
 - Use `list(existing_list)` or slicing `[:]` to create a new list.
 - The new list has the same elements.
 - Changing one list does not change the other.
- Note: This does not prevent problems with nested mutable objects: a *shallow copy* copies the references to the nested objects rather than the nested objects themselves.
- We will see how to handle this problem

```
a = [1, 2, 3]

b = list(a)
c = a[:]

b[0] = 99
c[0] = 100

print(a)    # [1, 2, 3]
```

This is a shallow copy.

Lecture 7

Lists and Indexes

Lecture 7 - Lists

- Out-of-range errors
 - Valid indexes go from 0 to `len(list)-1`.
 - Accessing a non-existing element raises `IndexError`.
 - This is one of the most common list errors.

```
values = [2.3, 4.5, 7.2]
```

```
values[0]    # OK
```

```
values[2]    # OK
```

```
values[3]    # IndexError
```

Always reason about the valid index range.

Lecture 7 - Lists

- `len()` with lists
 - `len(values)` returns the number of elements
 - The last valid index is `len(values) - 1`
 - Use `len()` instead of hard-coding the length
- Tips: use `len()` to check the validity of the list/string (or any iterable object) before apply operations; ask: is it the length I am expecting? is it an empty value?

```
values = [2.3, 4.5, 7.2]

n = len(values)      # 3
last = values[n-1]   # 7.2
```

`len()` adapts when the list changes.

Lecture 7 - Lists

Two uses of square brackets

```
values = [4]  
  
# creates a list  
# with one element
```

Indexing operator

```
values = [10, 20, 30]  
  
values[1]  
# selects the second element
```

[] can create a list or access an element (as for strings) depending on context.

Lecture 7 - Lists

- Negative indexes
 - `values[-1]` selects the last element.
 - `values[-2]` selects the second-to-last element.
 - `values[-len(values)]` selects the first element.
 - Useful when the position is known from the end.

list	'H'	12	16	9	1.3
Index	0	1	2	3	4
Index	-5	-4	-3	-2	-1

```
values = [10, 20, 30, 40]
```

```
values[-1]    # 40  
values[-2]    # 30  
values[-4]    # 10
```

Same rule already seen for strings.

Lecture 7 - Lists

- Slicing lists
 - `l[start:stop:step]`
 - Slicing extracts a portion of a list
 - The **stop** index is excluded
 - A **step** can be used, including a negative step
 - Slicing returns a **new list**.

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

```
values[1:4]    # [20, 30, 40]
```

```
values[:3]     # [10, 20, 30]
```

```
values[::2]    # [10, 30, 50]
```

```
values[::-1]   # [50, 40, 30, 20, 10]
```

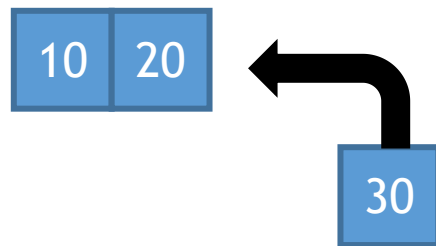
Same slicing syntax as strings.

List Operations

Adding, inserting, finding and removing elements

Lecture 7 - List Operations

- Adding elements with `.append()`
 - `.append()` is a method, not a function
 - Often we do not know all values in advance
→ Create an **empty list** `[]` and append values as they arrive.
 - `.append()` adds one element **at the end**.

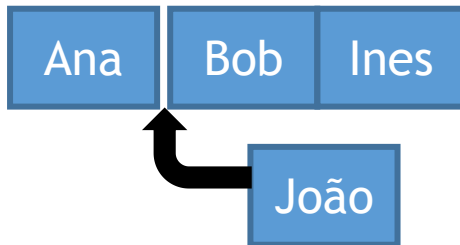


```
values = []  
  
values.append(10)  
values.append(20)  
values.append(30)  
  
print(values)  
# [10, 20, 30]
```

`.append()` **modifies the list.**

Lecture 7 - List Operations

- Inserting elements with `.insert()`
 - `.insert()` is a method, not a function
 - `.insert(index, value)` adds an element **at a specific position**.
 - Elements from that position move to the right.



```
names = ["Ana", "Bob", "Ines"]
names.insert(1, "João")

print(names)
# ["Ana", "João", "Bob", "Ines"]
```

Use when the position matters.

Lecture 7 - List Operations

- Membership with `in` / `not in`
 - Use `in` to test whether a value is present.
 - The result is a Boolean value.
 - Very useful inside `if` statements.

```
names = ["Ana", "Bob", "Carla"]  
  
if "Bob" in names:  
    print("Found")  
else:  
    print("Not found")
```

`in` works for strings and lists.

Lecture 7 - List Operations

- Finding the position of an element
 - method `.index()` returns the first matching position.
 - If the value is missing, it raises `ValueError`.
→ **Check membership** first when unsure.

```
names = ["Ana", "Bob", "Ana"]  
  
if "Bob" in names:  
    pos = names.index("Bob")  
    print(pos)          # 1
```

`index()` finds the first occurrence.

Lecture 7 - List Operations

- Two ways to remove an element: by value or by index
 - `.remove(value)` removes the first occurrence of value in the list
 - `.pop(index)` **returns** the elements at the specified position and **removes** it from the list

`remove(value)`

```
names = ["Ana", "Bob", "Carla"]  
  
names.remove("Bob")  
# removes by value  
  
print(names) # ["Ana", "Carla"]
```

`pop(index)`

```
names = ["Ana", "Bob", "Carla"]  
  
removed = names.pop(1)  
# removes by position  
# and returns the value  
print(names) # ["Ana", "Carla"]  
print(removed) # ["Bob"]
```

Both methods modify the list.

Lecture 7 - List Operations

- Useful built-in functions
 - `sum(values)` adds numeric elements
 - `min(values)` and `max(values)` inspect extremes
 - `sorted(values)` returns a new sorted list.
 - The original list is not modified by `sorted()`.

```
values = [12, 8, 15]

sum(values)      # 35
min(values)      # 8
max(values)      # 15
sorted(values)   # [8, 12, 15]
```

These functions also support common algorithms.

Lecture 7 - List Operations

- Useful built-in functions
 - `sum(values)` adds numeric elements.
 - `min(values)` and `max(values)` inspect extremes.
 - `sorted(values)` returns a new sorted list.
 - The original list is not modified by `sorted()`.

Be careful using these functions: list items must be **numeric** or at least *homogeneous*.

```
values = [12, 8, 15]

sum(values)      # 35
min(values)      # 8
max(values)      # 15
sorted(values)   # [8, 12, 15]
```

These functions also support common algorithms.

Lecture 7 - List Operations

- Useful built-in functions
 - `sum(values)` adds numeric elements.
 - `min(values)` and `max(values)` inspect extremes.
 - `sorted(values)` returns a new sorted list.
 - The original list is not modified by `sorted()`.

Be careful using these functions: list items must be **numeric** or at least *homogeneous*.

Some of them may be useful with **strings**, for instance, we may use `sorted(list_of_strings)` to sort in alphabetical order

```
values = ['Ana', 'Carla', 'Bob']  
  
sum(values)           # TypeError  
  
min(values)           # 'Ana'  
max(values)           # 'Carla'  
  
sorted(values)  
# ['Ana', 'Bob', 'Carla']
```

These functions also support common algorithms.

Lecture 7 - Lists

Concatenation with +

- The `+` operator creates a new list by concatenating two lists.
- The original lists are **not modified**.
- Both operands must be **lists**.

#common error:

```
sales + [30]      # OK
sales + 30        # TypeError
```

```
morning = [12, 15, 9]
afternoon = [20, 18]

sales = morning + afternoon

print(sales)      # [12, 15, 9, 20, 18]
print(morning)    # [12, 15, 9]
```

list + list → new list

list + value → TypeError

Lecture 7 - Lists

Replication with *

- The `*` operator repeats the elements of a list (same of strings)
- It is useful for quickly creating simple patterns or initial values.
- Be careful: it repeats **references** when elements are mutable (see next slide)

```
row = [0] * 4          # useful
table = [row] * 3      # warning: nested lists
```

```
zeros = [0] * 5
print(zeros)      # [0, 0, 0, 0, 0]

pattern = ["A", "B"] * 3
print(pattern)    # ['A', 'B', 'A', 'B', 'A', 'B']
```

Replication does not create a loop in your code, but it represents repeated elements.

Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



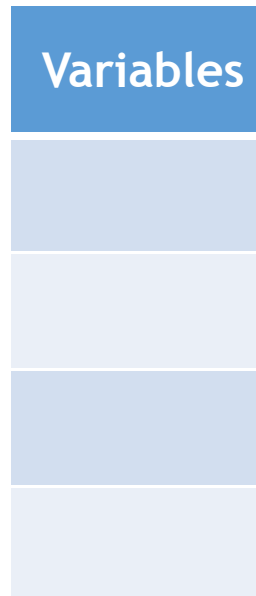
- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []
l2 = [l]*5          # [[], [], [], [], []]
l2[1].append('a')   # [['a'], ['a'], ['a'], ['a'], ['a']]

e1 = l2[2]
e1.append(1)

print(l2)
#[['a', 1], ['a', 1], ['a', 1], ['a', 1], ['a', 1]]

m = [[]]*5
m[1].append(10)
print(m)
#[[10], [10], [10], [10], [10]]
```



Address	Memory
0x1000	
0x1001	
0x1002	
0x1003	
0x1004	
0x1005	
...	
0x100C	

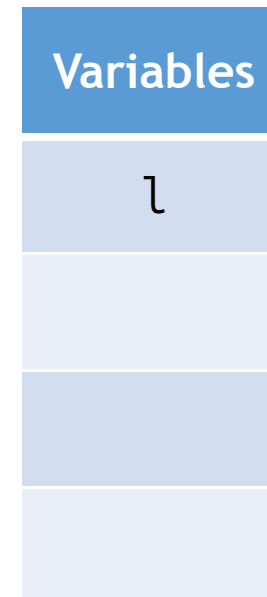
Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []  
l2 = [l]*5          # [[], [], [], [], []]  
l2[1].append('a')   # [['a'], [], [], [], []]  
  
e1 = l2[2]  
e1.append(1)  
  
print(l2)  
#[['a', 1], [], [], [], []]  
  
m = [[]]*5  
m[1].append(10)  
print(m)  
#[[10], [], [], [], []]
```



Address	Memory
0x1000	[]
0x1001	
0x1002	
0x1003	
0x1004	
0x1005	
...	
0x100C	

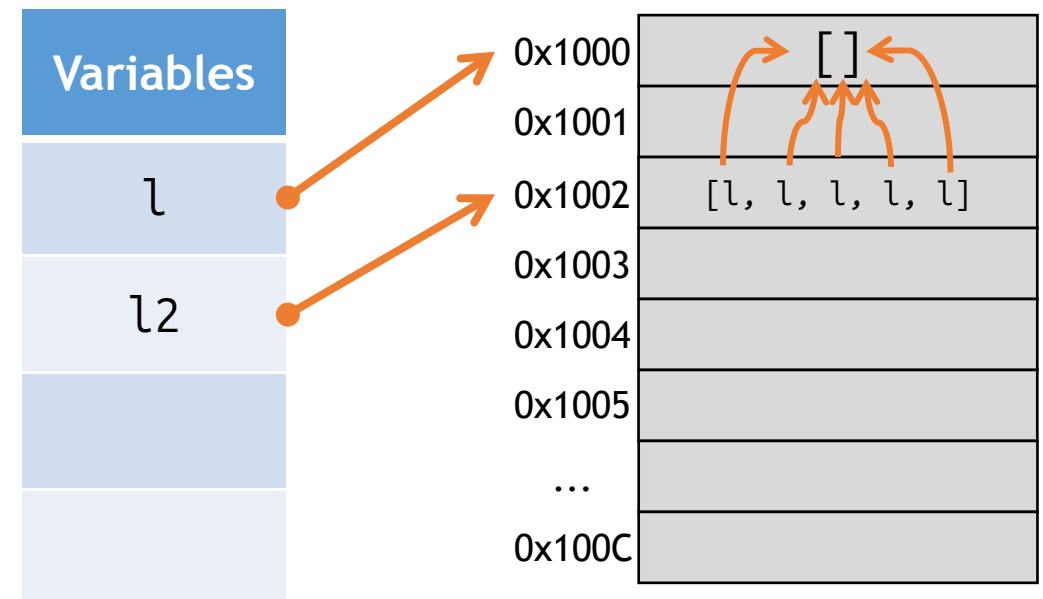
Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []  
l2 = [l]*5          # [[], [], [], [], []]  
l2[1].append('a')   # [['a'], [], [], [], []]  
  
e1 = l2[2]  
e1.append(1)  
  
print(l2)  
#[['a', 1], [], [], [], []]  
  
m = [[]]*5  
m[1].append(10)  
print(m)  
#[[10], [], [], [], []]
```



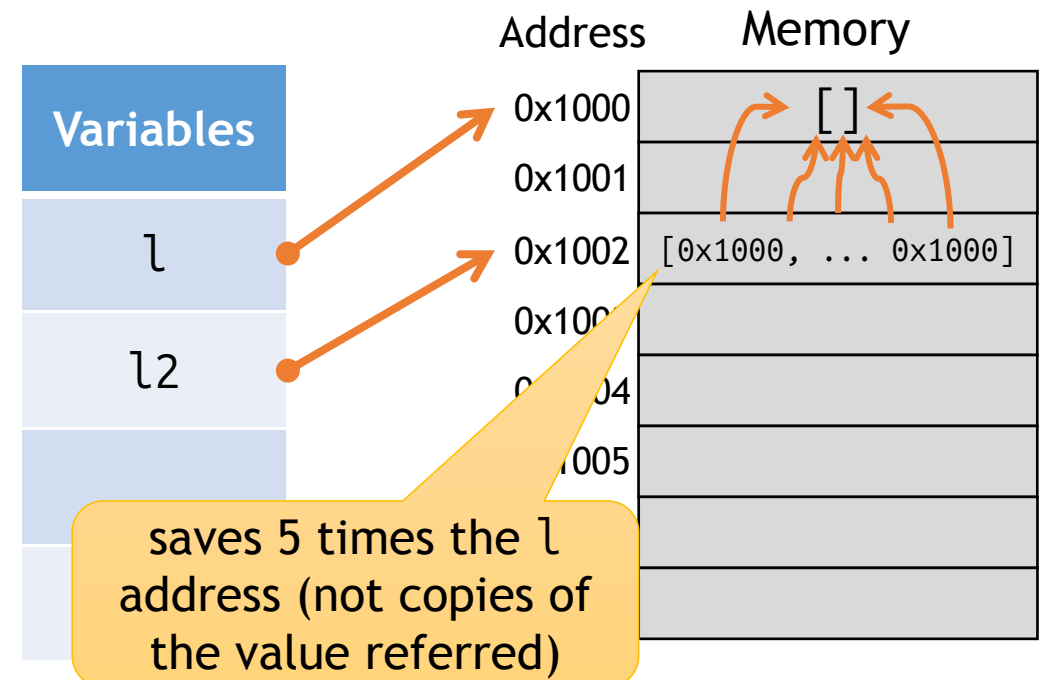
Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []  
l2 = [l]*5          # [[], [], [], [], []]  
l2[1].append('a')   # [['a'], ['a'], ['a'], ['a'], ['a']]  
  
e1 = l2[2]  
e1.append(1)  
  
print(l2)  
#[['a', 1], ['a', 1], ['a', 1], ['a', 1], ['a', 1]]  
  
m = [[]]*5  
m[1].append(10)  
print(m)  
#[[10], [10], [10], [10], [10]]
```



Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



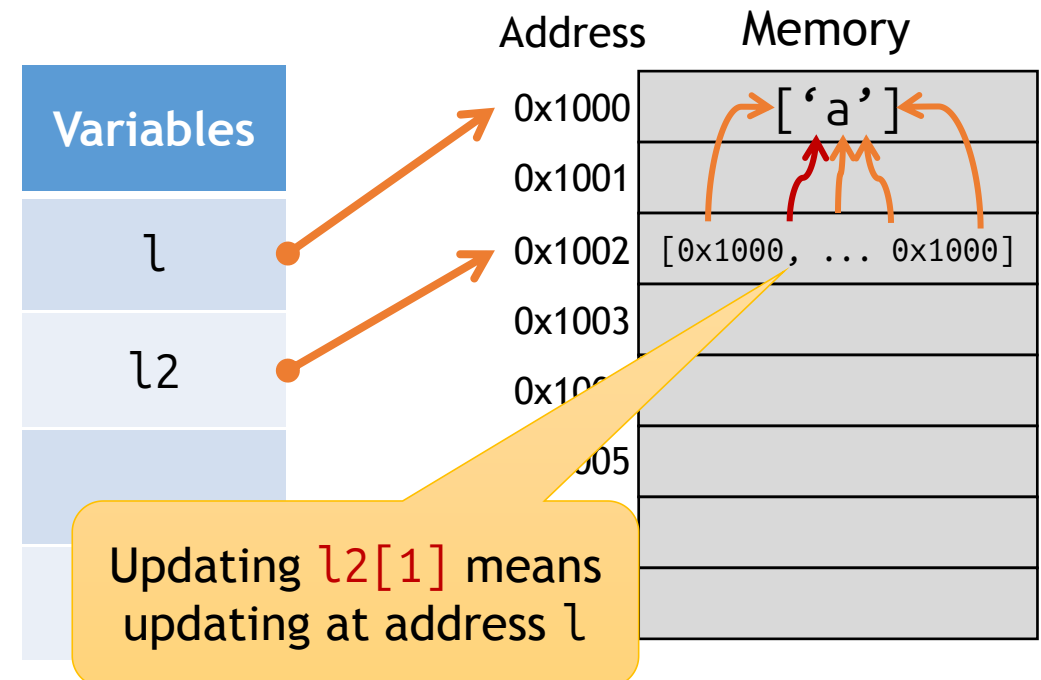
- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []
l2 = [l]*5          # [[], [], [], [], []]
l2[1].append('a')   # [['a'], [], [], [], []]

e1 = l2[2]
e1.append(1)

print(l2)
#[['a', 1], [], [], [], []]

m = [[]]*5
m[1].append(10)
print(m)
#[[10], [], [], [], []]
```



Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



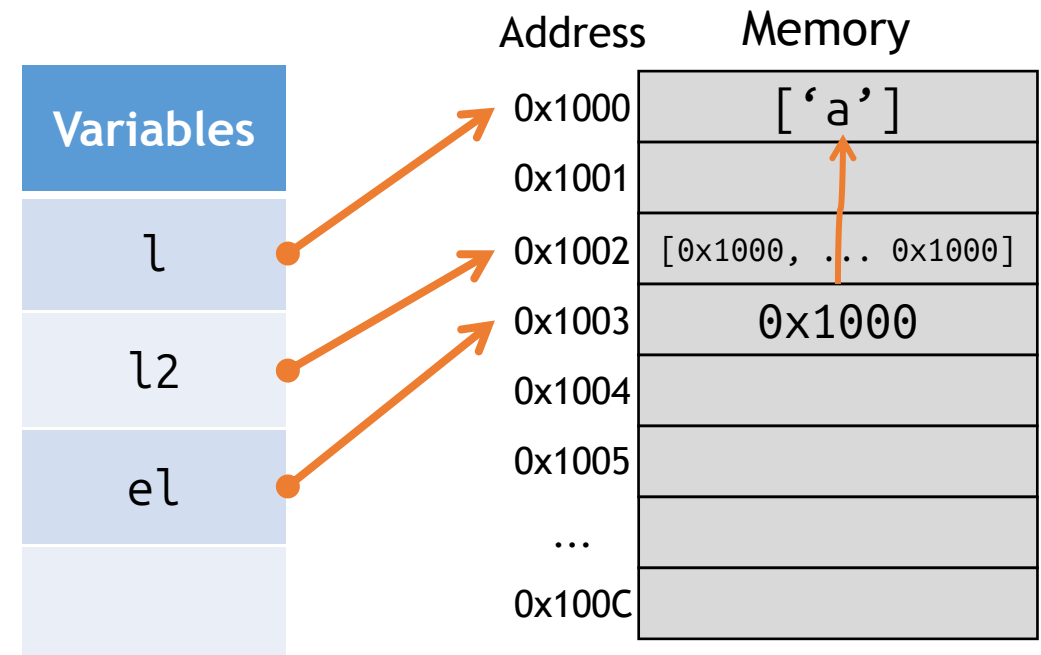
- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []
l2 = [l]*5          # [[], [], [], [], []]
l2[1].append('a')   # [['a'], ['a'], ['a'], ['a'], ['a']]

el = l2[2]
el.append(1)

print(l2)
#[['a', 1], ['a', 1], ['a', 1], ['a', 1], ['a', 1]]

m = [[]]*5
m[1].append(10)
print(m)
#[[10], [10], [10], [10], [10]]
```



Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



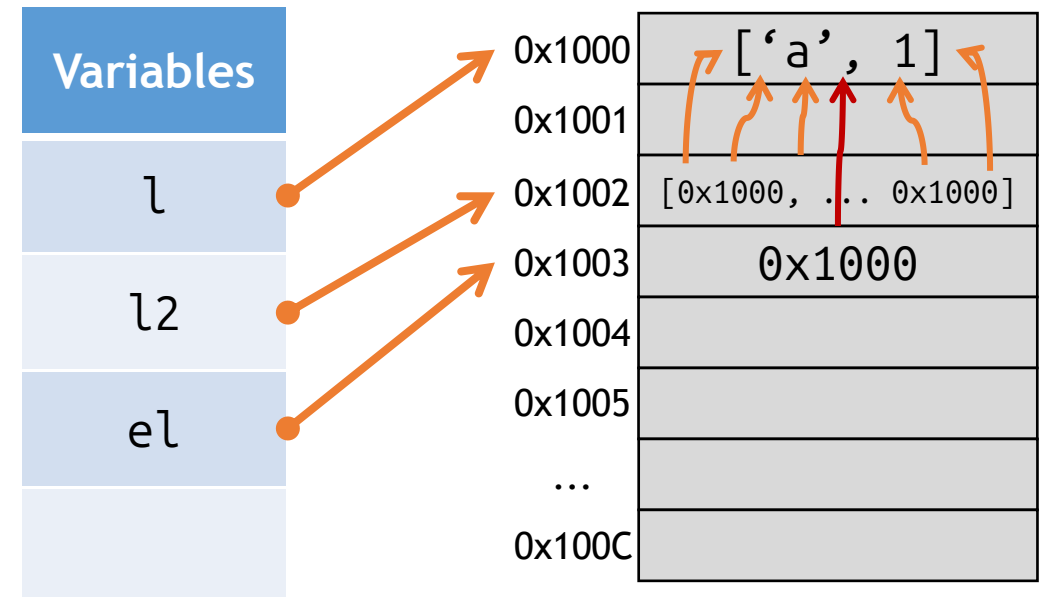
- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []
l2 = [l]*5          # [[], [], [], [], []]
l2[1].append('a')   # [['a'], [], [], [], []]

el = l2[2]
el.append(1)

print(l2)
#[['a', 1], ['a', 1], ['a', 1], ['a', 1], ['a', 1]]

m = [[]]*5
m[1].append(10)
print(m)
#[[10], [10], [10], [10], [10]]
```



Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



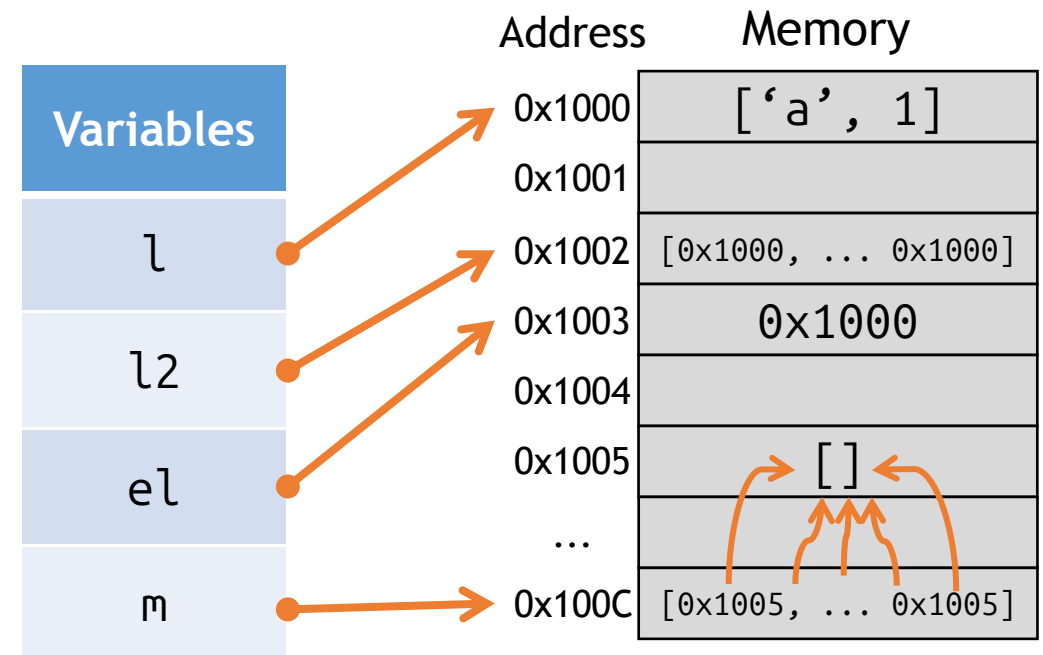
- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []
l2 = [l]*5          # [[], [], [], [], []]
l2[1].append('a')   # [['a'], [], [], [], []]

el = l2[2]
el.append(1)

print(l2)
#[['a', 1], ['a', 1], ['a', 1], ['a', 1], ['a', 1]]

→ m = [[]]*5
m[1].append(10)
print(m)
#[[10], [10], [10], [10], [10]]
```



Lecture 7 - Lists

Common error with nested mutables (part 2 of 3634943298478...)



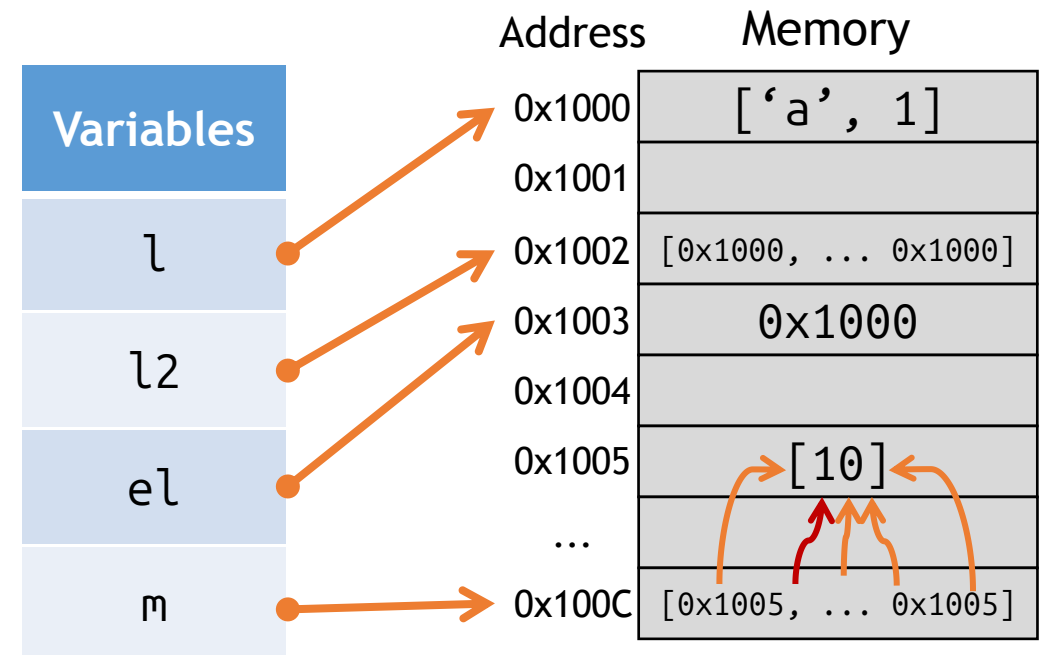
- Be careful using `*` with mutables: it repeats references to the same object
- often, not what you intended to do
 - example:

```
l = []
l2 = [l]*5          # [[], [], [], [], []]
l2[1].append('a')   # [['a'], ['a'], ['a'], ['a'], ['a']]

el = l2[2]
el.append(1)

print(l2)
#[['a', 1], ['a', 1], ['a', 1], ['a', 1], ['a', 1]]

m = [[]]*5
m[1].append(10)
print(m)
#[[10], [10], [10], [10], [10]]
```



Lecture 7 - Lists

Adding many elements: .extend()

- .append() adds one element at the end of a list.
- .extend() adds all elements from another iterable.
- Unlike +, .extend() **modifies the existing list**.

append(x): **add x as one element**

```
queue = ["Ana", "Bob"]
new_arrivals = ["Carla", "Daniel"]

queue.append(new_arrivals)
# ['Ana', 'Bob', ['Carla', 'Daniel']]

queue = ["Ana", "Bob"]
queue.extend(new_arrivals)
# ['Ana', 'Bob', 'Carla', 'Daniel']
```

extend(iter): **add each element of iter**

Lecture 7 - Lists

Combining list operations

Common ways to combine or add elements:

Operation	Effect	Modifies original?
<code>a + b</code>	creates a new concatenated list	No
<code>a * n</code>	creates a new repeated list	No
<code>a.append(x)</code>	adds x as one element	Yes
<code>a.extend(b)</code>	adds each element of b	Yes

Question to ask: do I need a new list, or do I want to modify the existing one?

documentation and examples about list methods: <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Lecture 7 - Lists

Equality and difference

- Lists can be compared for equality and difference.
- Two lists are **equal** if they have:
 - the same **length**
 - the same **elements**
 - in the same **order**
- An empty list [], as a boolean, is considered **False** (exactly as an empty string)

```
bool([]) == False # True!
```

```
l = []  
if not l:  
    print("empty")
```

```
[1, 2, 3] == [1, 2, 3]    # True  
[1, 2, 3] == [3, 2, 1]   # False  
[1, 2]      == [1, 2, 0]  # False  
  
[1, 2, 3] != [3, 2, 1]    # True
```

Order matters for lists.

Lecture 7 - Lists

Ordering comparisons: < and >

- Python can compare lists **lexicographically**.
- It compares element by element from left to right.
- The first different element determines the result.
- This is similar to string comparison.
- Also <= and >= exist

```
[1, 2, 3] < [1, 2, 4]    # True  
[1, 3]    > [1, 2, 9]    # True  
[1, 2]    < [1, 2, 0]    # True
```

[1, 2, 3]

[1, 2, 4]

same

same

3 < 4

Use ordering comparisons only when they match the meaning of your data.

Lecture 7 - Lists and Strings

From string to list: `split()`

- The string method `split()` breaks a string into pieces.
- The result is a **list of strings**.
- By default, it splits on **whitespace**.
- A different separator can be specified as an argument.

```
text = "milk bread apples"
items = text.split()
# ['milk', 'bread', 'apples']

line = "Ana,18,Economics"
fields = line.split(",")
# ['Ana', '18', 'Economics']
```

`split()` is very common when reading simple structured text.

<https://docs.python.org/3.14/builtins/stdtypes.html>

Lecture 7 - Lists and Strings

From list to string: `.join()`

- `.join()` combines strings using a separator.
- Syntax may look unusual at first:
 - `separator.join(sequence_of_strings)`
- All elements must be strings.

```
numbers = [1, 2, 3]
', '.join(numbers)      # TypeError
```

```
items = ['milk', 'bread', 'apples']

text = ', '.join(items)
print(text)
# milk, bread, apples

code = '-'.join(['LIS', '2026', 'A7'])
# LIS-2026-A7
```

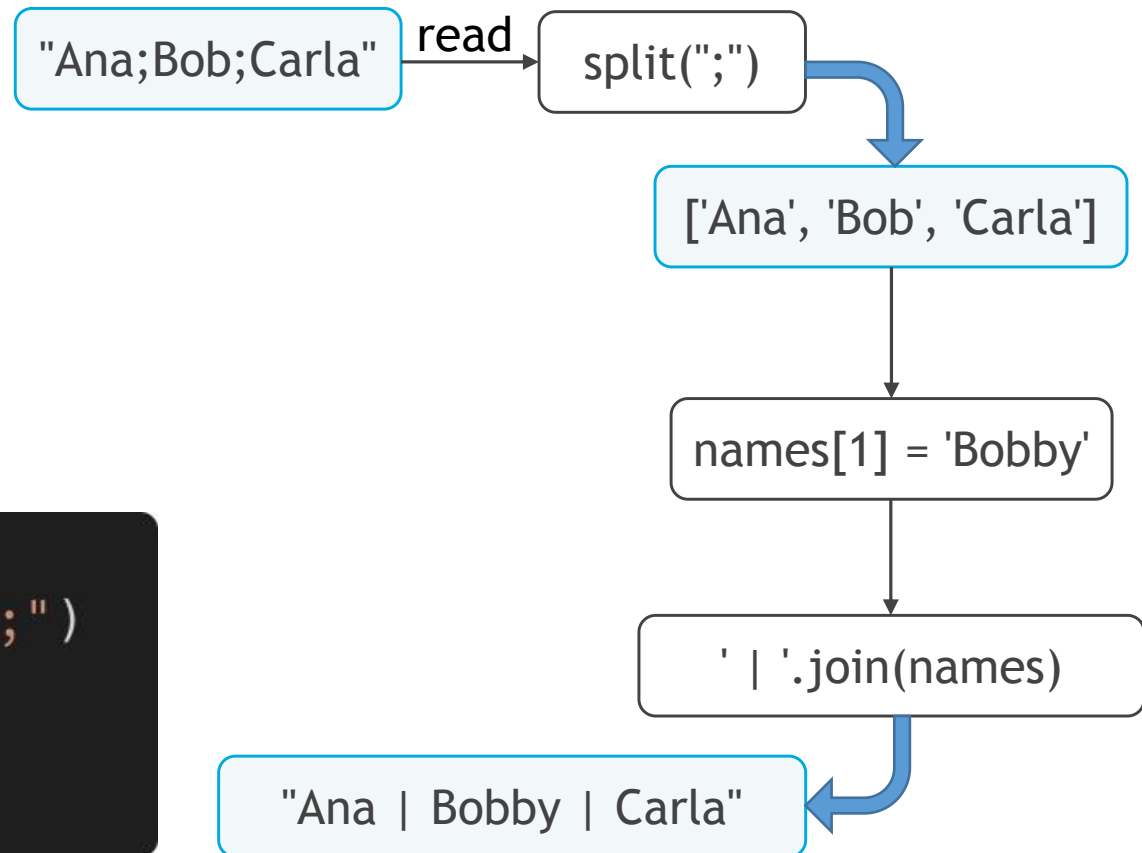
Think of the separator as the glue
between the elements.

Lecture 7 - Lists and Strings

`split()` and `join()` together

- A common text-processing workflow:
 1. split a string into parts
 2. process or inspect the list
 3. join the parts back into a string

```
names = "Ana;Bob;Carla".split(";")  
names[1] = "Bobby"  
result = " | ".join(names)
```



Lecture 7 - Lists

Swapping elements

- Sometimes we need to exchange two values.
- Python supports multiple assignment:
 - `a, b = b, a`
- This is useful for **swapping** list elements too.

No temporary variable is needed.

```
a = 10
b = 20

a, b = b, a

print(a)    # 20
print(b)    # 10
```

```
values = [4, 9, 2]

values[0], values[2] = values[2], values[0]

print(values)    # [2, 9, 4]
```

Lecture 7 - Recap

Additional list operations

Tool	Use
<code>+</code>	concatenate two lists into a new list
<code>*</code>	replicate elements
<code>extend()</code>	add all elements from another iterable
<code>==, !=</code>	compare list contents
<code><, >, <=, >=</code>	lexicographical ordering comparison
<code>.split()</code>	string $\rightarrow$ list of strings
<code>sep.join()</code>	list of strings $\rightarrow$ string
<code>a, b = b, a</code>	swap values
<code>[expr for x in it]</code>	build a list compactly

These tools extend what loops can do with sequences.

Lists and Loops

Processing many values

Lecture 7 - Lists and Loops

- As for strings, we have 2 ways to loop over lists (and other sequences):
 - Loop over **indexes** → `range(len(list))` to obtain a sequence from 0 to `len(list)-1`
 - Loop over **elements**
 - Usually, a `for _ in loop`

Loop over indexes

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

for i in range(len(values)):
    print(i, values[i])
```

Loop over elements

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

for value in values:
    print(value)
```

Use indexes only when you need positions.

Lecture 7 - Lists and Loops

- Indexes and values together
 - `enumerate()` gives **both index and element**.
 - By default, counting starts from 0.
 - Use `start=1` only when you want human numbering.

```
names = ["Ana", "Bob", "Carla"]  
  
for i, name in enumerate(names):  
    print(i, name)  
  
for pos, name in enumerate(names, start=1):  
    print(pos, name)
```

Useful for reports and tables.

Lecture 7 - Lists and Loops

- Building a list with a loop
 - Start from an **empty list** []
 - Read or compute one value at each iteration.
 - **Append** each value to the list.
 - After the loop, process the complete list.

```
values = []  
  
for i in range(5):  
    text = input("Value: ")  
    value = float(text)  
    values.append(value)  
  
print(values)
```

A common pattern in data processing.

Lecture 7 - Lists and Loops

List comprehension

- A *list comprehension* is a compact way to build a new list (initialization).
- It combines:
 - an expression
 - a for loop
 - optionally, a condition
- Use it when it remains readable.

```
squares = []  
  
for x in range(1, 6):  
    squares.append(x ** 2)  
  
print(squares)  
# [1, 4, 9, 16, 25]
```

```
squares = [x ** 2 for x in range(1, 6)]  
  
print(squares)  
# [1, 4, 9, 16, 25]
```

Same result, more compact syntax.

Lecture 7 - Lists and Loops

List comprehension syntax

Basic form:

```
[expression for variable in iterable]
```

With a filter condition:

```
[expression for variable in iterable if condition]
```

The **condition** decides which elements are included.

```
numbers = [3, -1, 0, 8, -5]

positive = [x for x in numbers if x > 0]
# [3, 8]

double_positive = [2 * x for x in numbers if x > 0]
# [6, 16]
```

Read it aloud: “take 2*x for each x in numbers if x is positive”.

Lecture 7 - Lists and Loops

Comprehensions in small data tasks

- Comprehensions are useful after `split()`.
- Example:
 - Read numbers typed on one line.
 - Each piece is initially a string and must be converted.
 - Find min and max prices

```
line = input("Enter prices separated by spaces: ")
parts = line.split()

prices = [float(x) for x in parts]

print(min(prices))
print(max(prices))
```

input → string → split → list of strings → list of numbers

Lecture 7 - Lists and Loops

When not to use a list comprehension

- A comprehension should make the code clearer, not more cryptic.
- Prefer an explicit loop when:
 - the logic is long or nested
 - you need several statements
 - you update several variables
 - you want to debug step by step

```
# Good: simple transformation  
names = [name.upper() for name in names]
```

```
# Better as a normal loop:  
# - read input  
# - validate  
# - update counters  
# - print messages
```

Compact is good only when it is still understandable.

Lecture 7 - Lists and Loops

- Common list algorithms
 - accumulate a **total**
 - **count** values satisfying a condition
 - find a **maximum** or **minimum**
 - **search** for a value
 - build a transformed list

```
count = 0

for value in values:
    if value >= 10:
        count = count + 1

print(count)
```

Loops + lists = the basis of many programs.