CSCI 141

Lecture 22
Variables are References
Mutability's Implications

Announcements

Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).

Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).
    - Suggestion: get it mostly done this week.

Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).
    - Suggestion: get it mostly done this week.
- Next week is Thanksgiving week

Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).
  - Suggestion: get it mostly done this week.
- Next week is Thanksgiving week
  - No labs

Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).
    - Suggestion: get it mostly done this week.
- Next week is Thanksgiving week
    - No labs
    - Class Monday

Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).
    - Suggestion: get it mostly done this week.
- Next week is Thanksgiving week
    - No labs
    - Class Monday
    - No class Wednesday or Friday

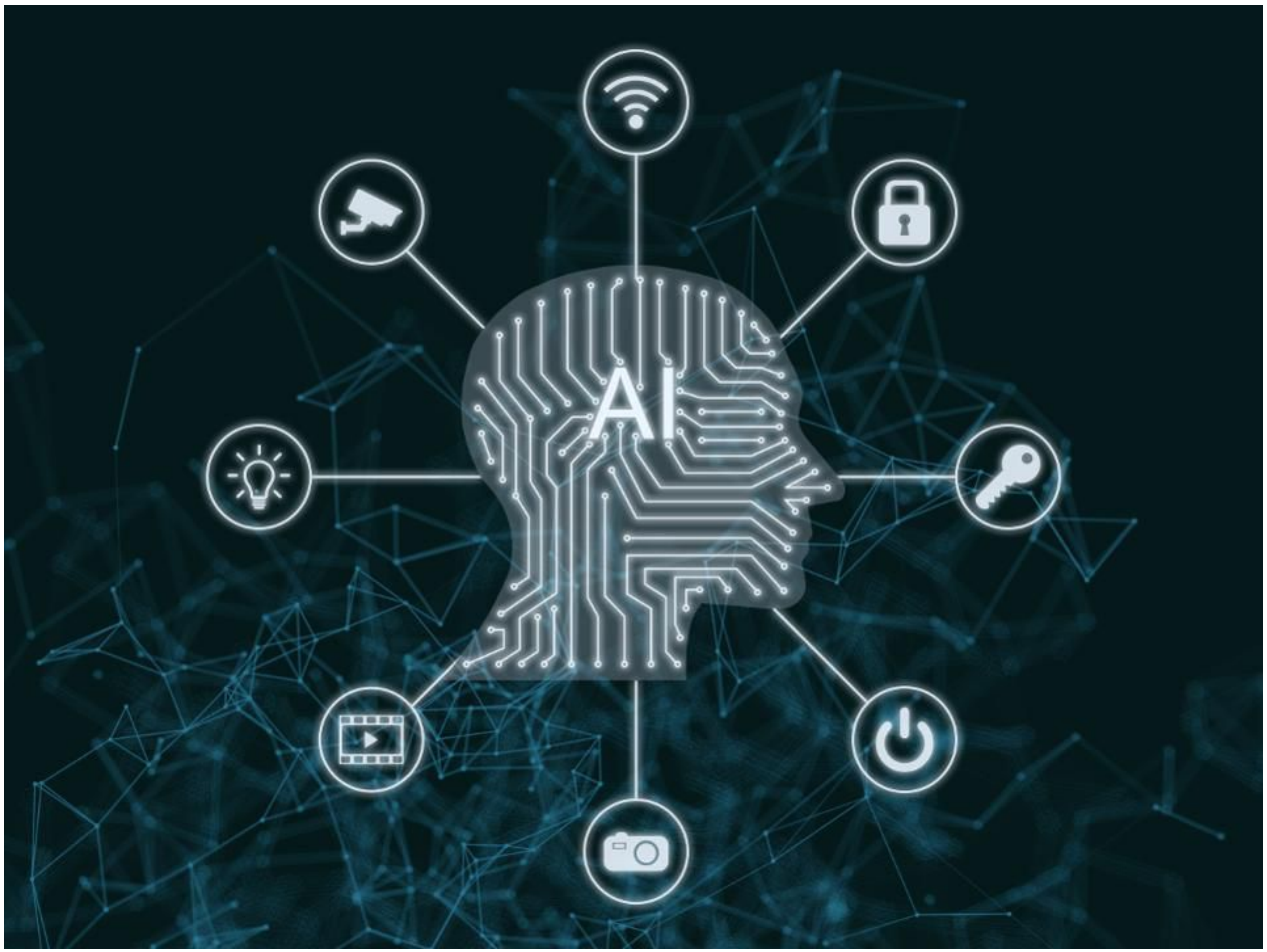
Goals

- Understand what you're being asked to do in A5.
- Understand the implications of variables holding references to mutable objects:
  - Multiple variables can refer to the same object.
- Be able to draw memory diagrams for code snippets involving mutable objects.
- Know how to query or modify lists using the following: `index, insert, remove, del`

A5

xkcd.com/1838

A5: Machine Learning!



A5: Machine Learning!

$$
\begin{aligned}
\mathbb{E}[\mathbf{x}] &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \int \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\} \mathbf{x}\,\mathrm{d}\mathbf{x} \\
&= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\} (\mathbf{z}+\boldsymbol{\mu})\,\mathrm{d}\mathbf{z} \qquad (2.58)
\end{aligned}
$$

A5: Machine Learning!

okay but it's not actually that crazy

Let's talk about **creatures**.

Some creatures are monsters.
Some creatures are not monsters.

You can't always tell by looking at them.

A5: Machine Learning!

okay but it's not actually that crazy

Let's talk about **creatures**.

Some creatures are monsters.
Some creatures are not monsters.

You can't always tell by looking at them.

Problem setup:
we have a dataset of known monsters and non-monsters.
and we want to look at their attributes to figure out how to decide whether a new, never-before-seen creature is a monster.

Known Creatures:

Size	Toothiness	Monster?
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters

2. Find the average **size** of monsters

3. Cast a "vote" as follows:

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters

2. Find the average **size** of monsters

3. Cast a "vote" as follows:

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters

   = 4

2. Find the average **size** of monsters

3. Cast a "vote" as follows:

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters

   = 4

2. Find the average **size** of monsters

3. Cast a "vote" as follows:

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters = 4

2. Find the average **size** of monsters = 15

3. Cast a "vote" as follows:

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters = 4

2. Find the average **size** of monsters = 15

3. Cast a "vote" as follows:

- If the unknown creature's **size** is closer to the Monster average, vote Monster.

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters = 4

2. Find the average **size** of monsters = 15

3. Cast a "vote" as follows:

- If the unknown creature's **size** is closer to the Monster average, vote Monster.

- If the creature's **size** is closer to the non-Monster average, vote non-Monster.

Known Creatures:

Size	Tooth	Mnstr
2	12	N
3	11	N
6	18	N
5	23	N
12	100	Y
21	84	Y
17	104	Y
10	112	Y

Unknown creature:

4	22	?

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

A5

- Creatures --> tumors
- Monster --> malignant
- Non-monster --> benign
- Size and toothiness --> radius, texture, area, ...
  (a total of 10 attributes)

I want to show you something weird.

I want to show you something weird.

- Demo:

```
a = [4, 5]

b = a

b[0] = 1

print(a[0])
```

Objects and Variables: Digging a little deeper

When we talked about variables...

Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

`number` | 2 |

Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:

Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:

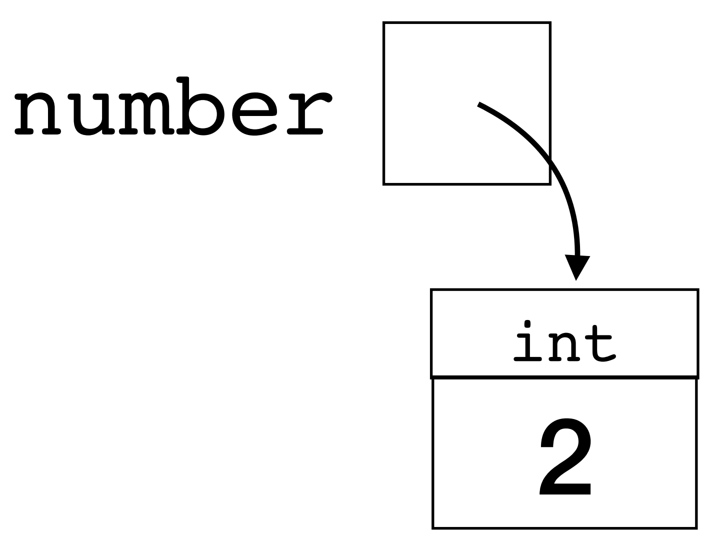
`number`

Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:

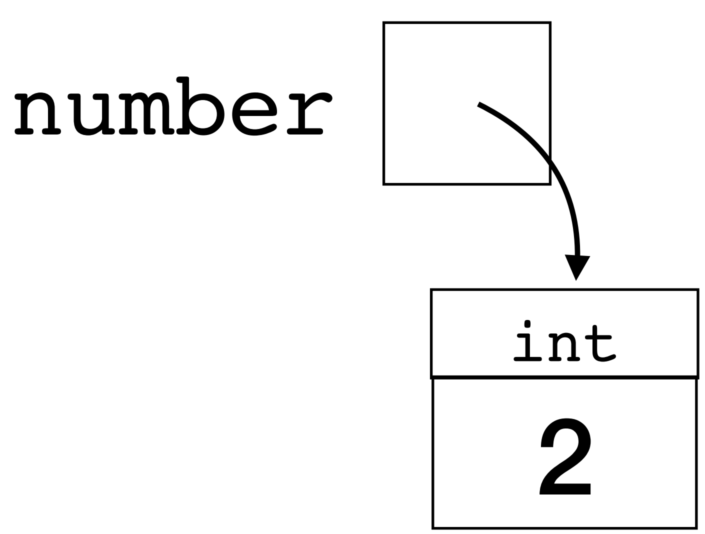


Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:



All variables store references to **objects**.

Objects can have any type

All variables store references to objects

In code:

In memory:

All variables store references to objects

In code:

In memory:

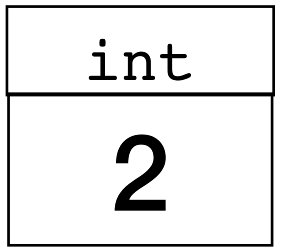
```
number = 2
```

All variables store references to objects

In code:

```
number = 2
```

In memory:

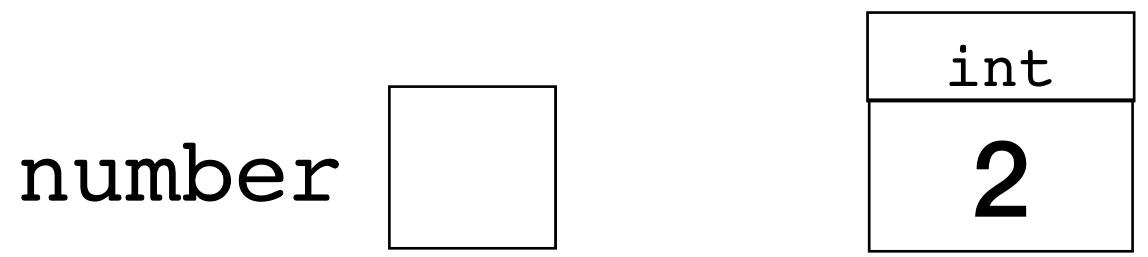


All variables store references to objects

In code:

```
number = 2
```

In memory:

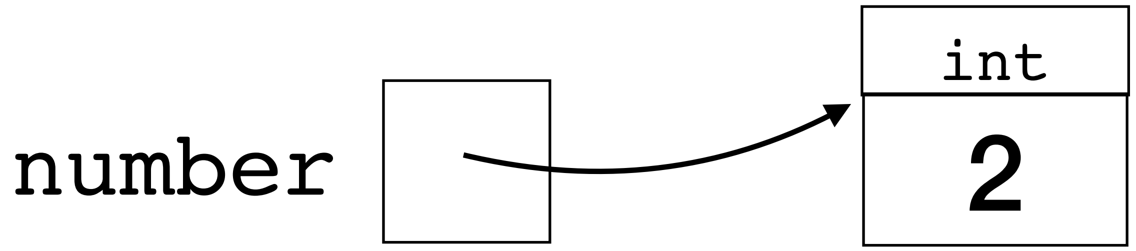


All variables store references to objects

In code:

```
number = 2
```

In memory:

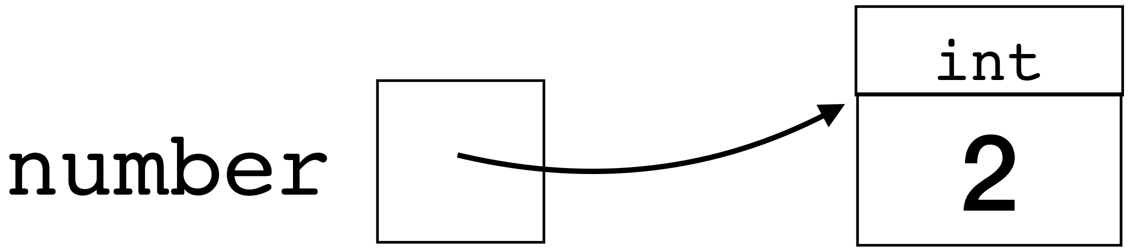


All variables store references to objects

In code:

```
number = 2
```

In memory:



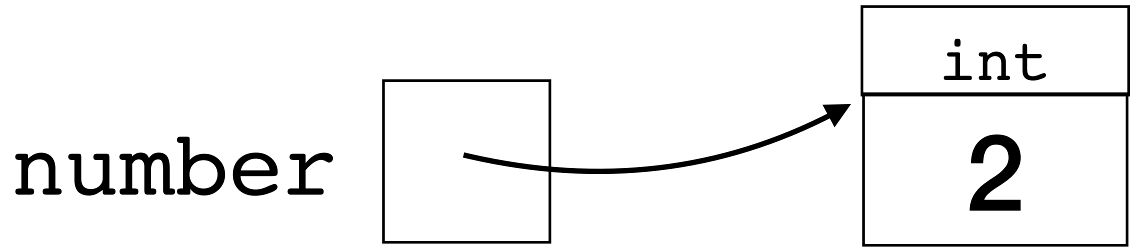
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



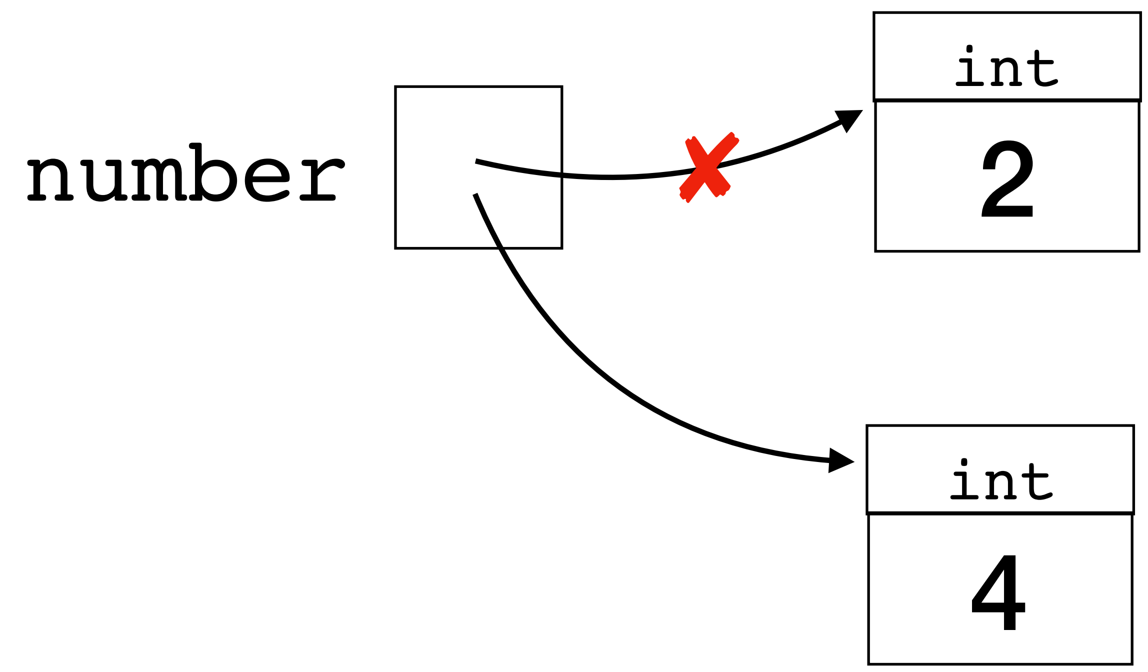
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



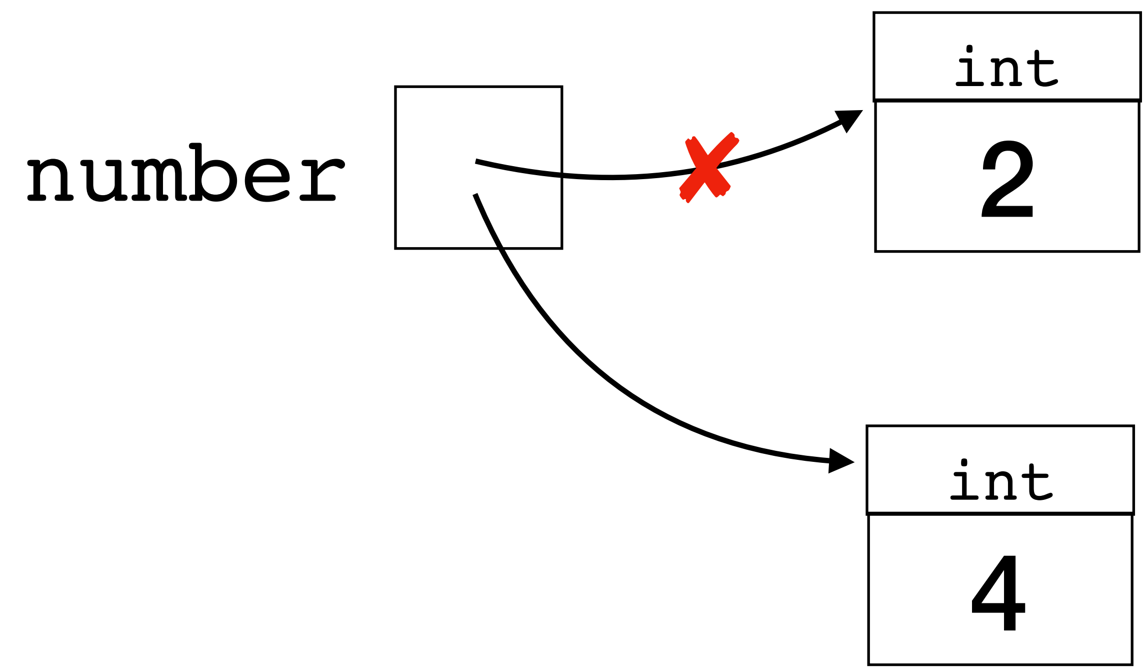
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



Like strings, `ints` are immutable:
You can't change its value.
You can only make a new one with a different value.

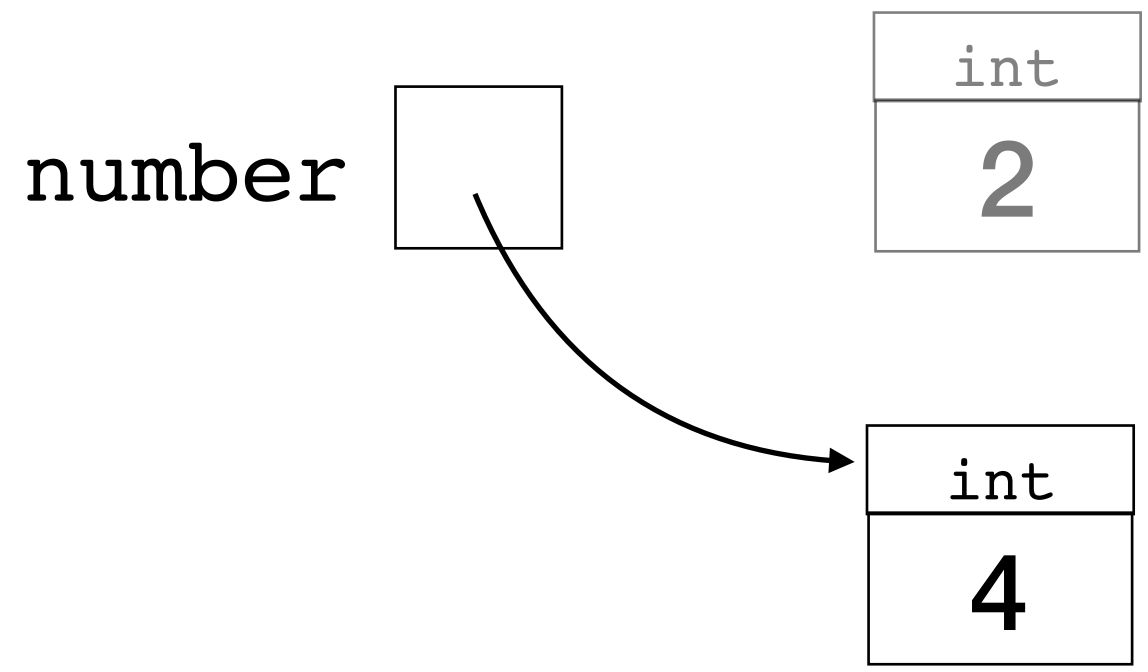
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



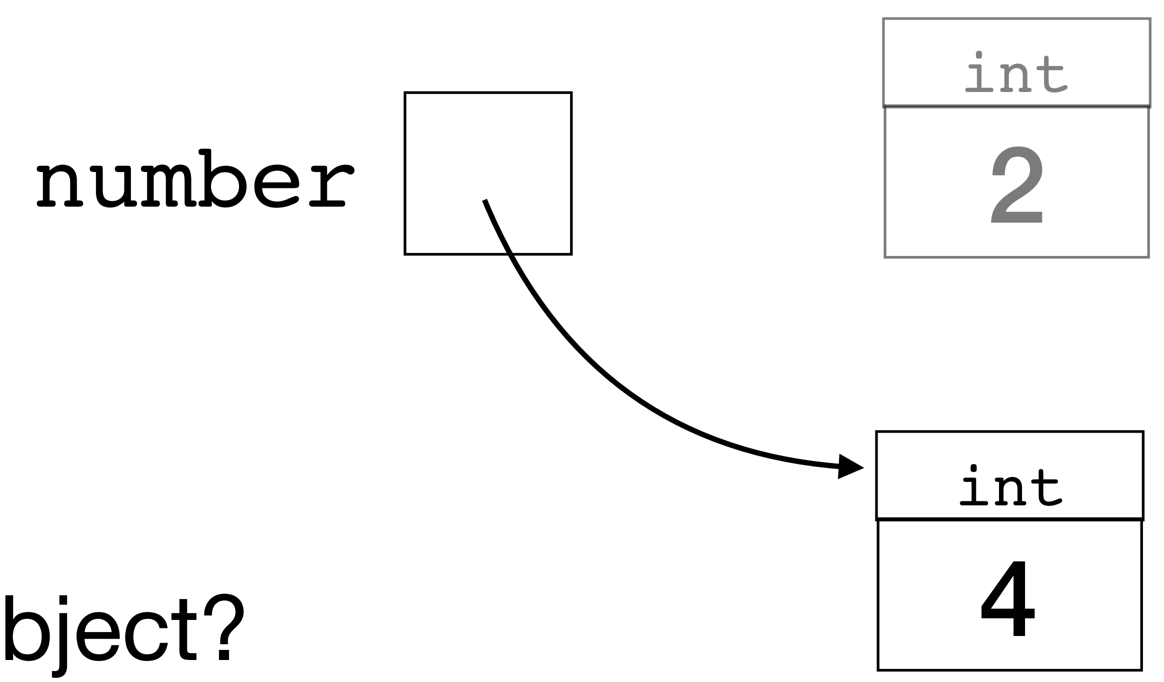
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



Aside: What happens to the 2 object?

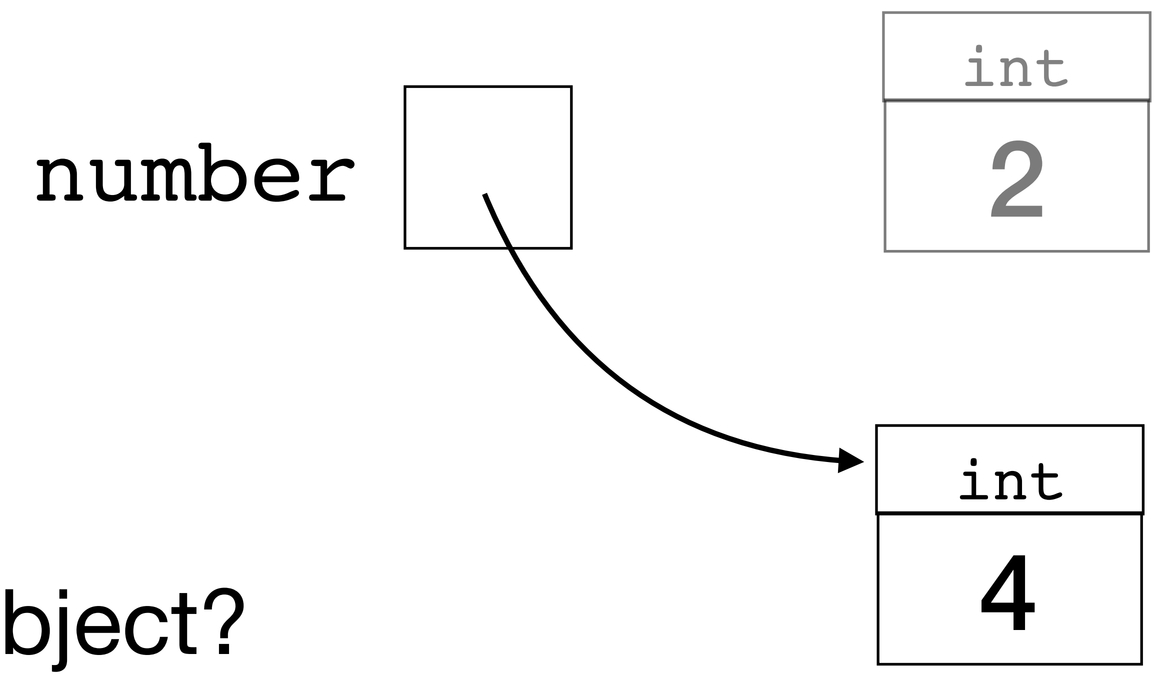
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



Aside: What happens to the 2 object?

- If no variables refer to it, Python deletes it automatically.

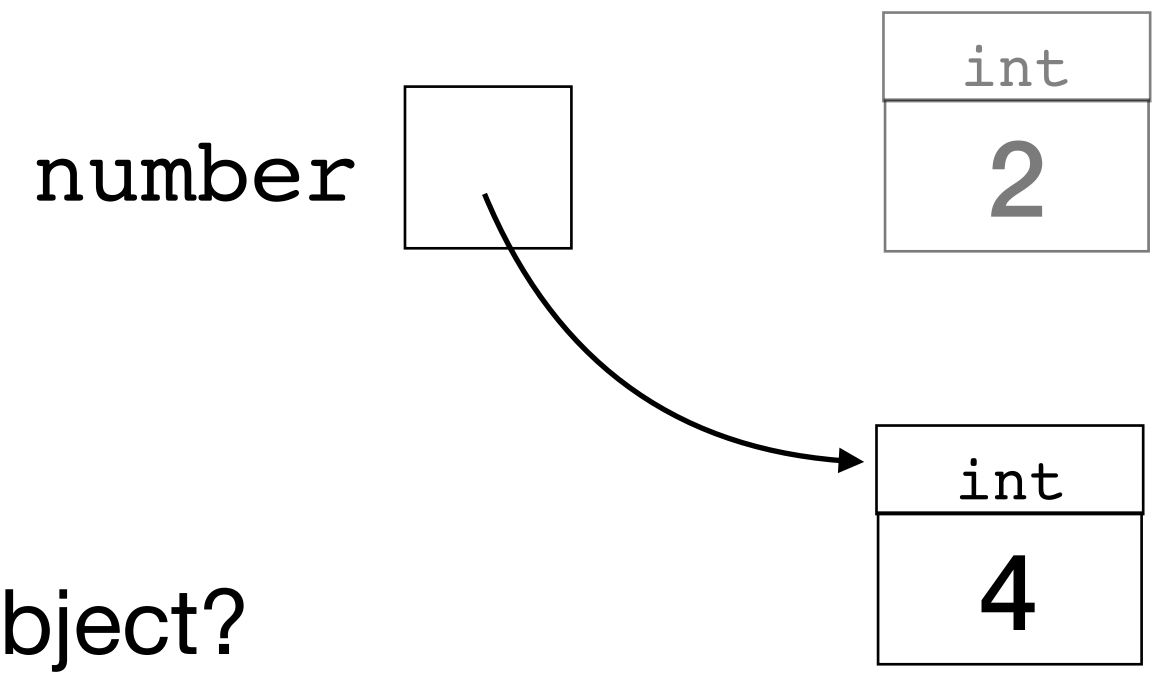
All variables store references to objects

In code:

```
number = 2


number = 4
```

In memory:



Aside: What happens to the 2 object?

- If no variables refer to it, Python deletes it automatically.
- This is called *garbage collection*.

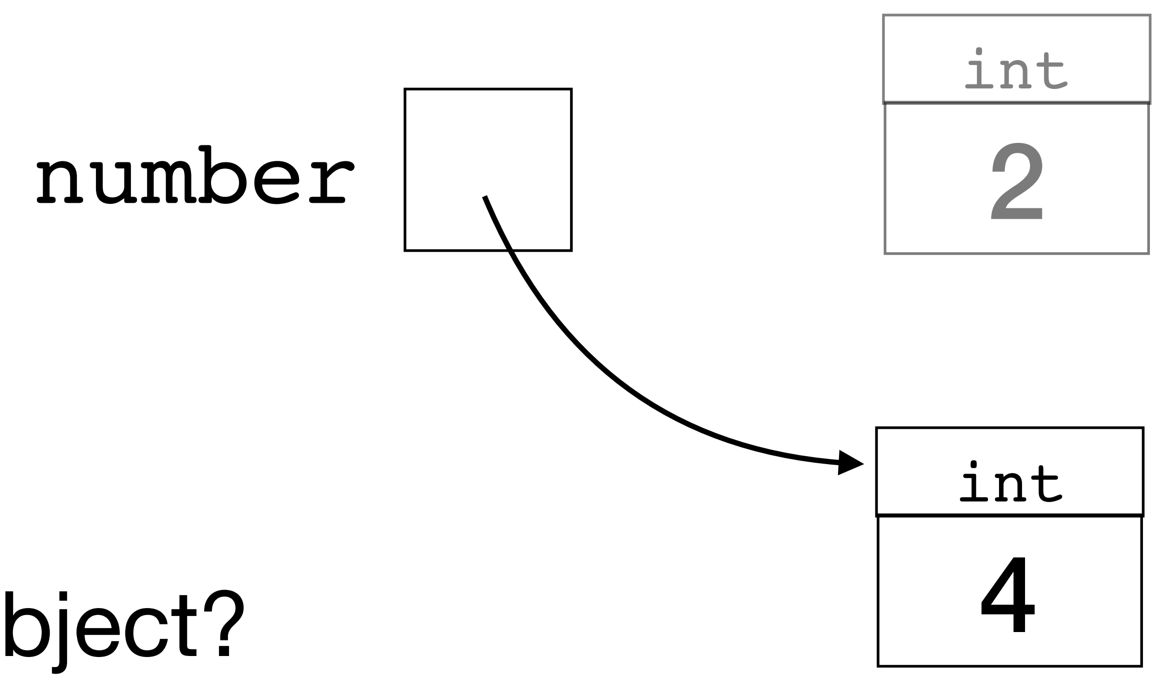
All variables store references to objects

In code:

```
number = 2

number = 4
```

In memory:



Aside: What happens to the 2 object?

- If no variables refer to it, Python deletes it automatically.
- This is called *garbage collection*.

For immutable objects, the fact that variables hold references doesn't have many interesting consequences.

Worksheet - Problem 1

Execute the following, drawing and updating the memory diagram for each variable and object involved.

```
number = 2
other_number = number
number += 1
```

Worksheet - Problem 1

Execute the following, drawing and updating the memory diagram for each variable and object involved.

```
number = 2
other_number = number
number += 1
```

(whiteboard)

All variables store references to objects

What about **mutable** objects?

In code:

In memory:

All variables store references to objects

What about **mutable** objects?

In code:

```
a = [4, 5]
```

In memory:

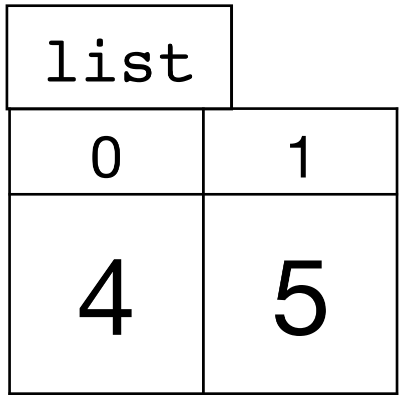
All variables store references to objects

What about **mutable** objects?

In code:

```
a = [4, 5]
```

In memory:



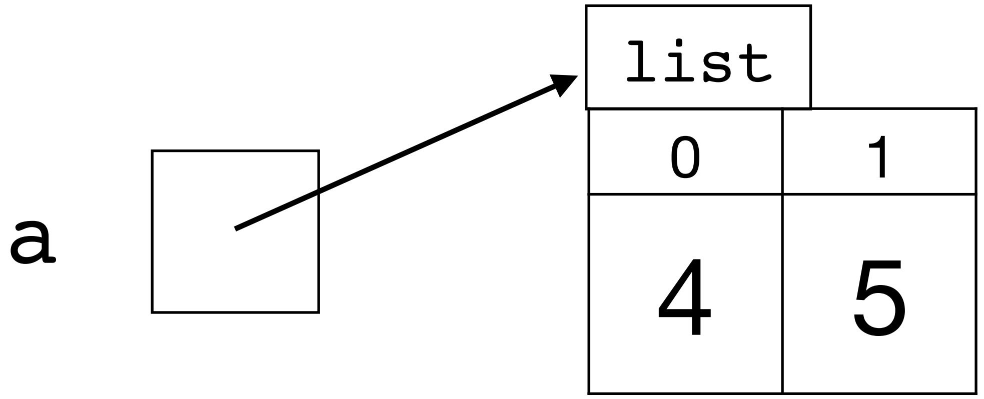
All variables store references to objects

What about **mutable** objects?

In code:

```
a = [4, 5]
```

In memory:



All variables store references to objects

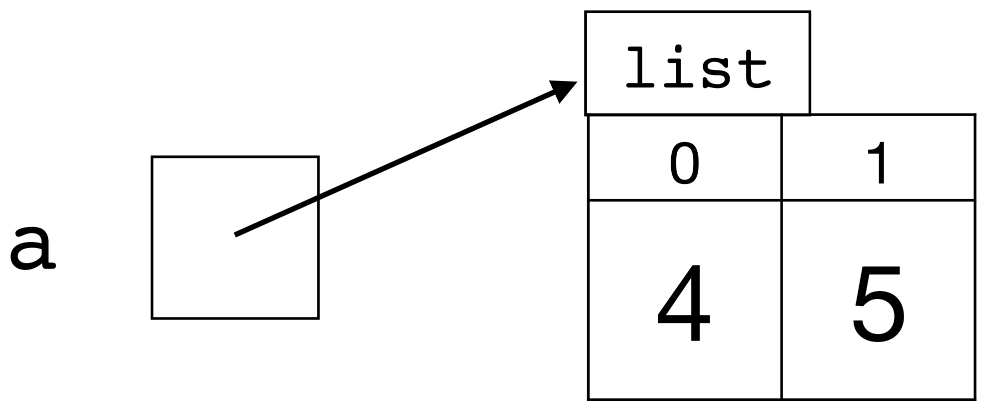
What about **mutable** objects?

In code:

```
a = [4, 5]

b = a
```

In memory:



All variables store references to objects

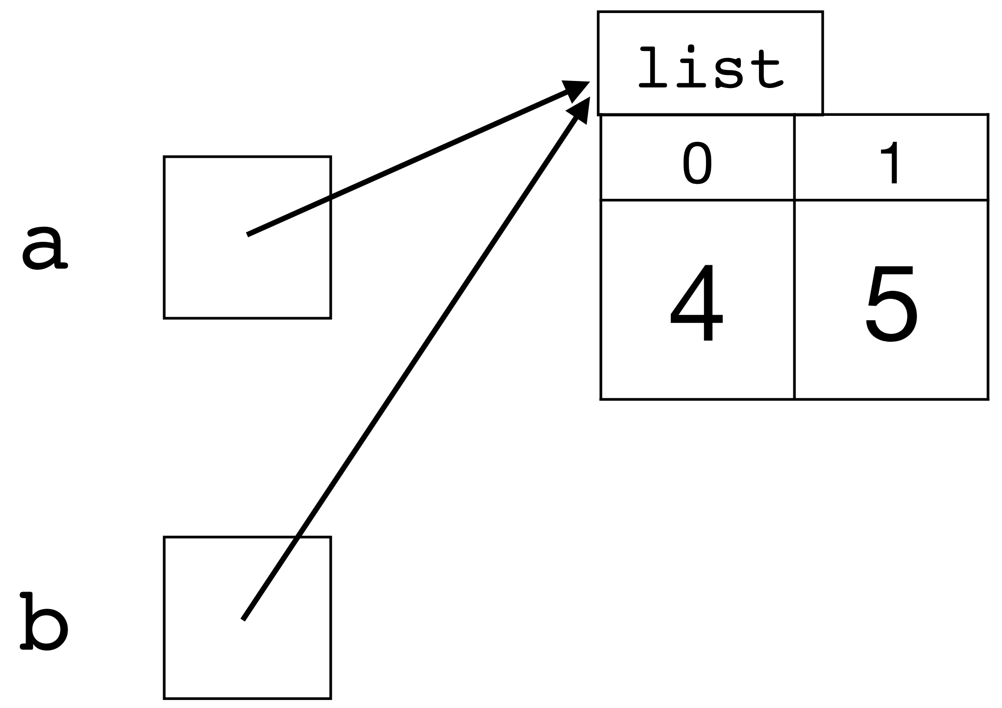
What about **mutable** objects?

In code:

```
a = [4, 5]

b = a
```

In memory:



All variables store references to objects

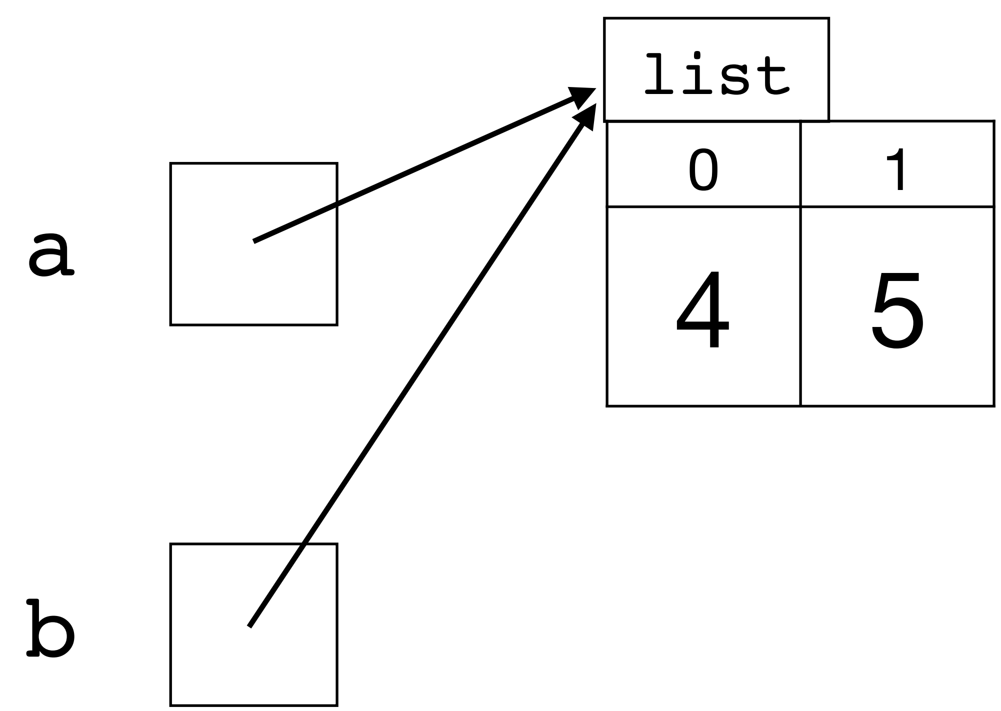
What about **mutable** objects?

In code:

```
a = [4, 5]

b = a
```

In memory:



The value of `a` is a *reference* to that list object, so the new value of `b` is also a *reference* to that **same** list!

All variables store references to objects

What about **mutable** objects?

In code:

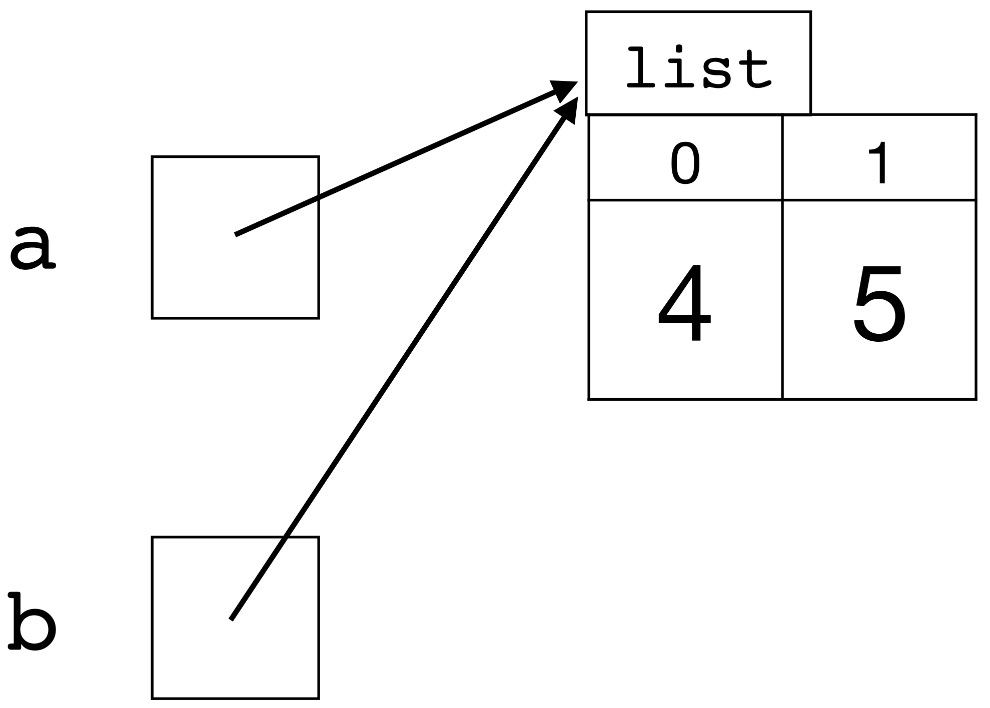
```
a = [4, 5]

b = a



[1, 5] # !!!
```

In memory:



All variables store references to objects

What about **mutable** objects?

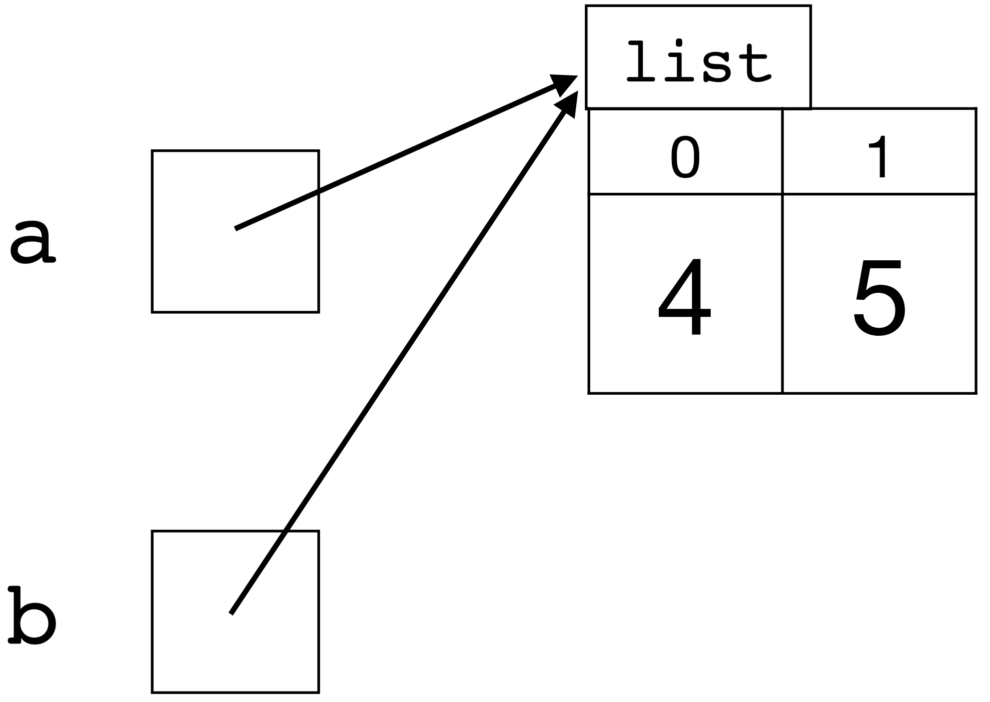
In code:

```
a = [4, 5]

b = a
b[0] = 1


[1, 5] # !!!
```

In memory:



All variables store **references** to objects

What about **mutable** objects?

In code:

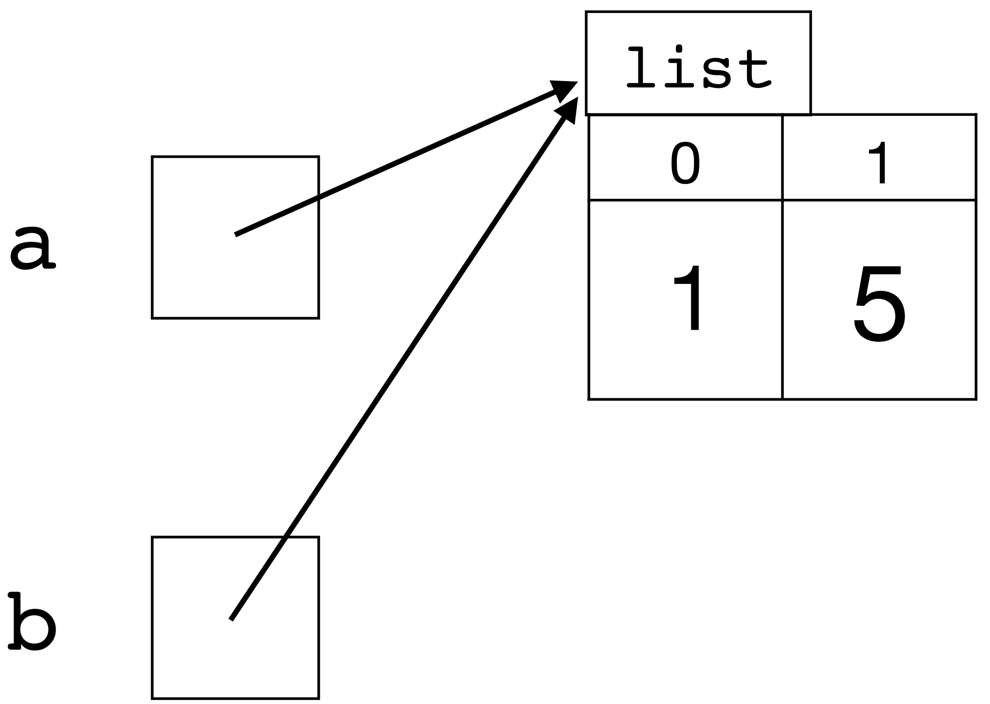
```
a = [4, 5]

b = a
b[0] = 1



[1, 5] # !!!
```

In memory:



All variables store references to objects

What about **mutable** objects?

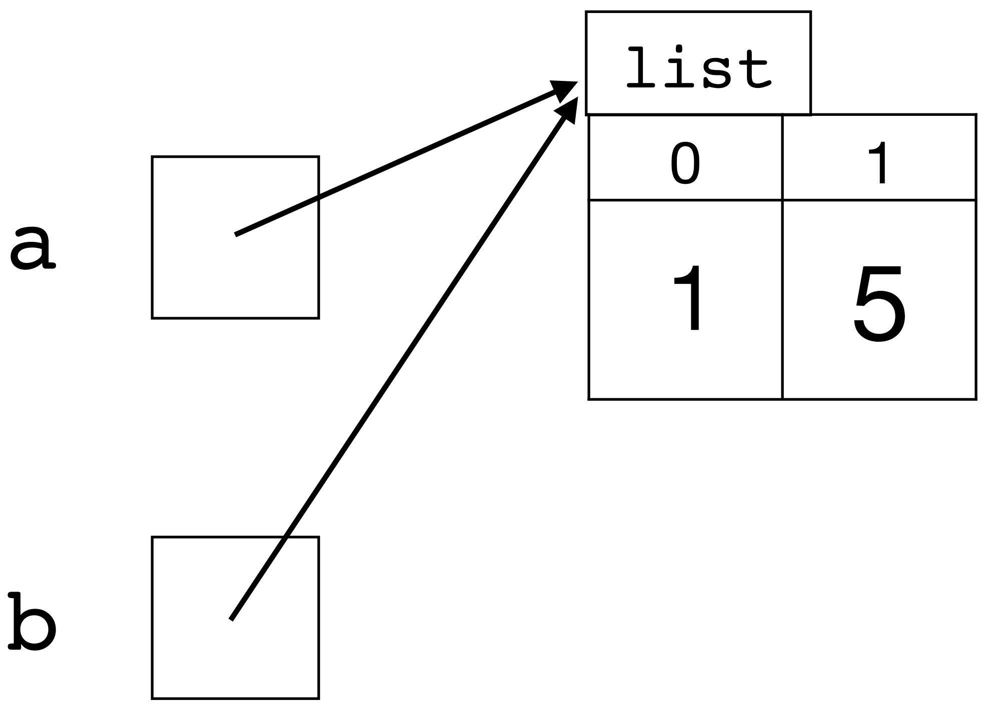
In code:

```
a = [4, 5]

b = a
b[0] = 1

print(a)
[1, 5] # !!!
```

In memory:



All variables store references to objects

What about **mutable** objects?

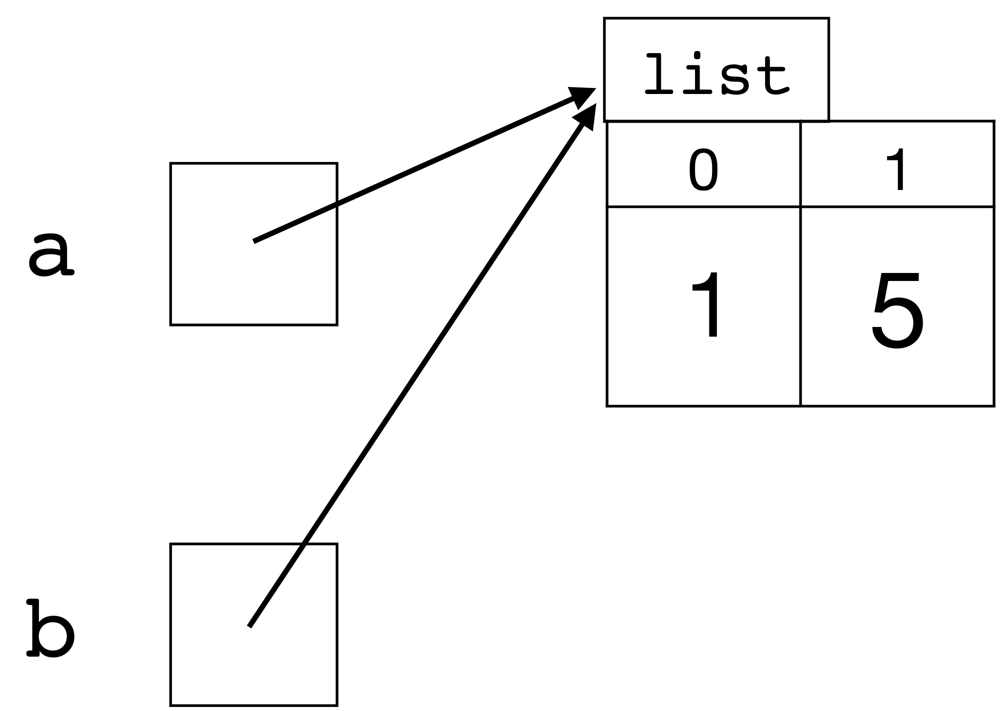
In code:

```
a = [4, 5]

b = a
b[0] = 1

print(a)
[1, 5] # !!!
```

In memory:



More than one variable can refer to the **same object.**

Don't make this mistake

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

you **did not** just create a copy of `a`

Don't make this mistake

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

you **did not** just create a copy of `a`

To create a true copy of a **mutable** object, you **can't** simply assign the object to a new variable.

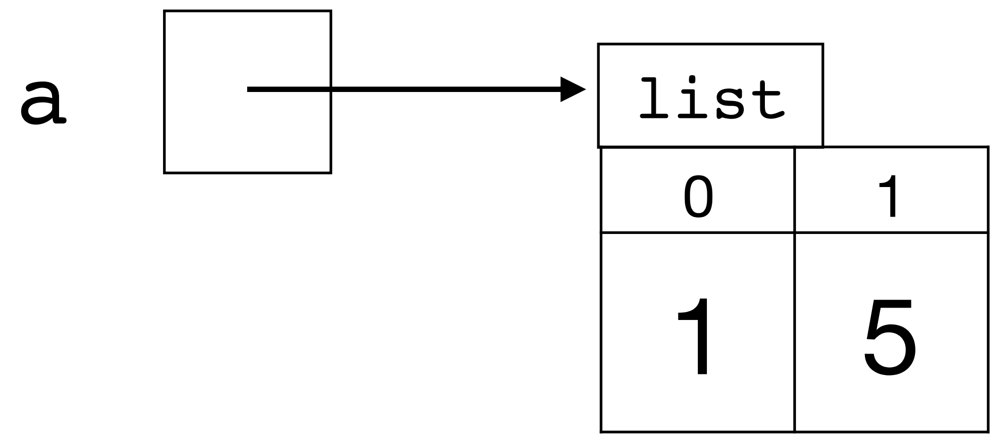
List elements store references to objects

List elements are just like variables!

In code:

```
a = [4, 5]
```

In memory:



I lied to you again!

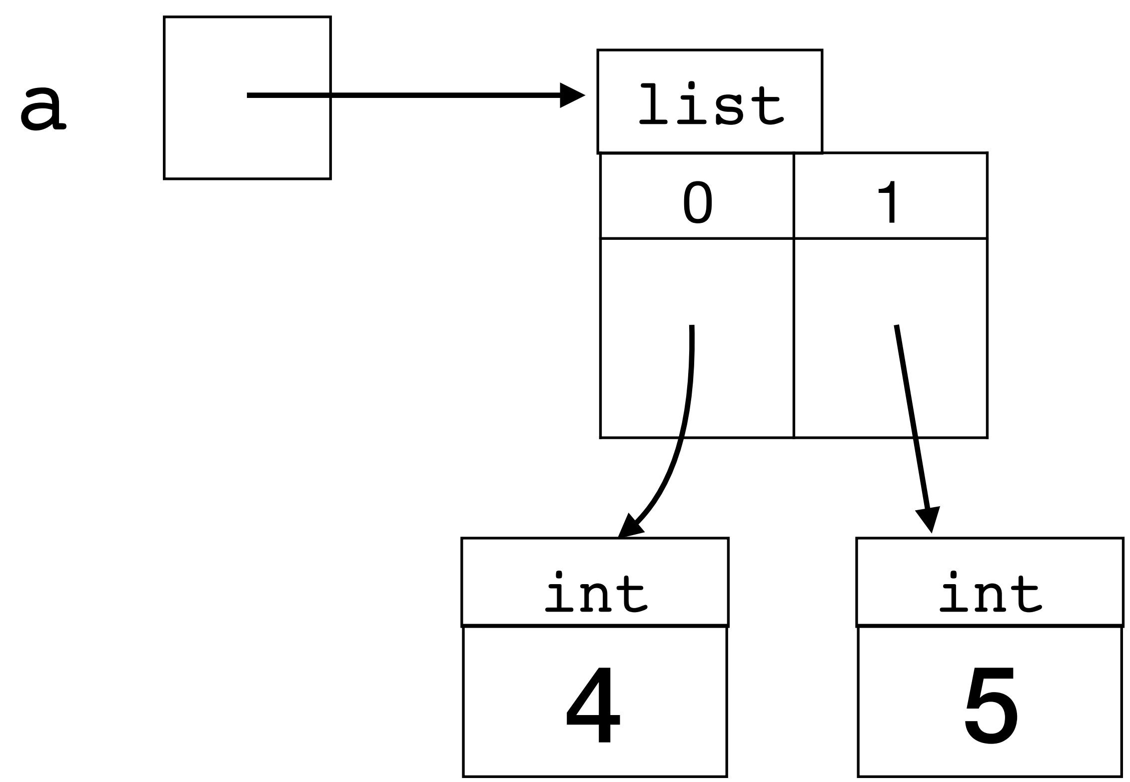
List elements store references to objects

List elements are just like variables!

In code:

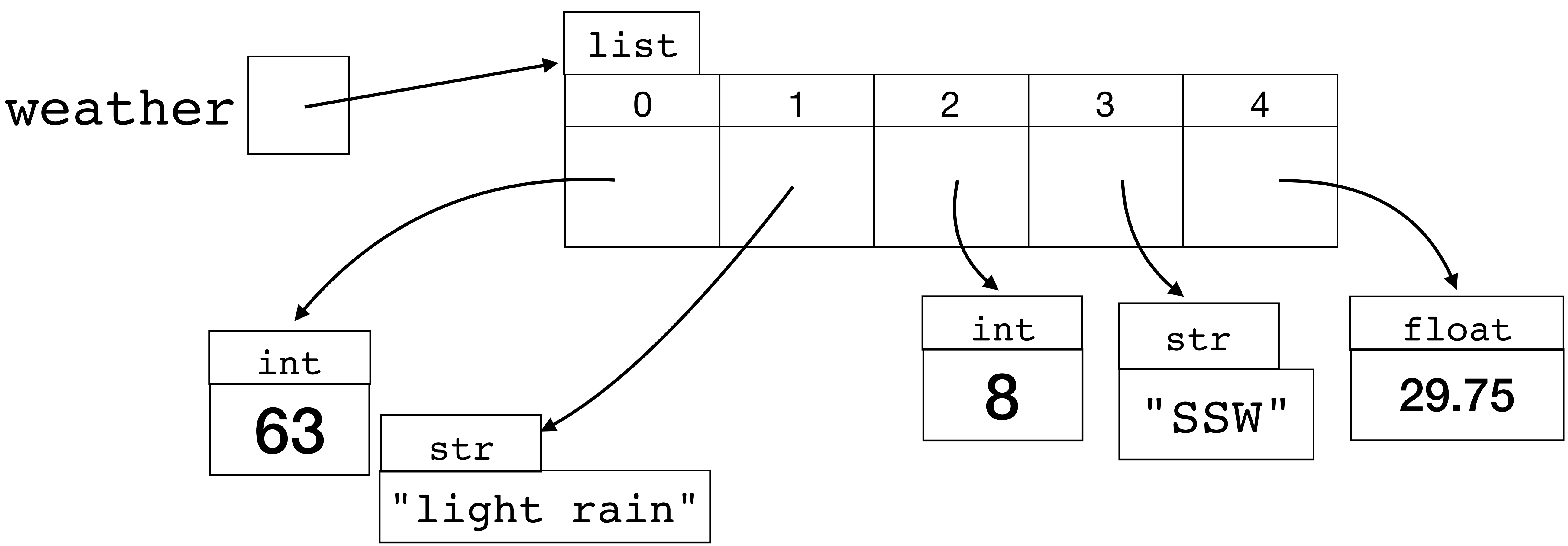
```
a = [4, 5]
```

In memory (the true picture):



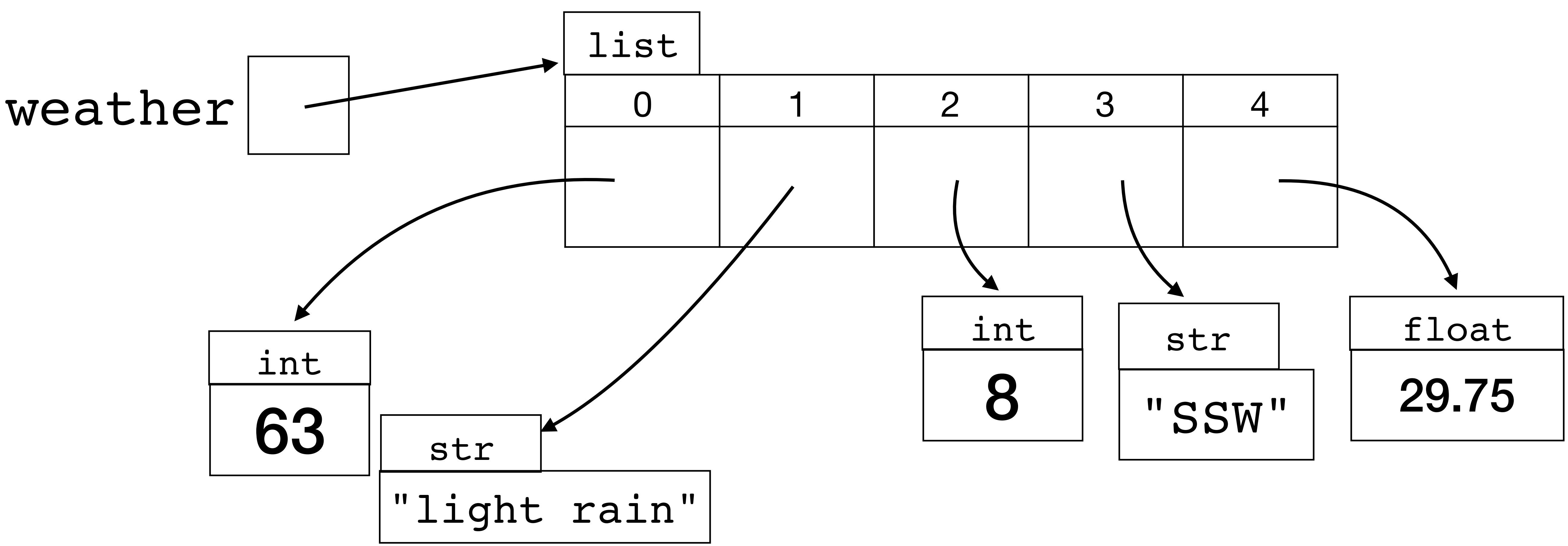
List elements store references to objects

```
weather = [63, "light rain", 8, "SSW", 29.75]
```



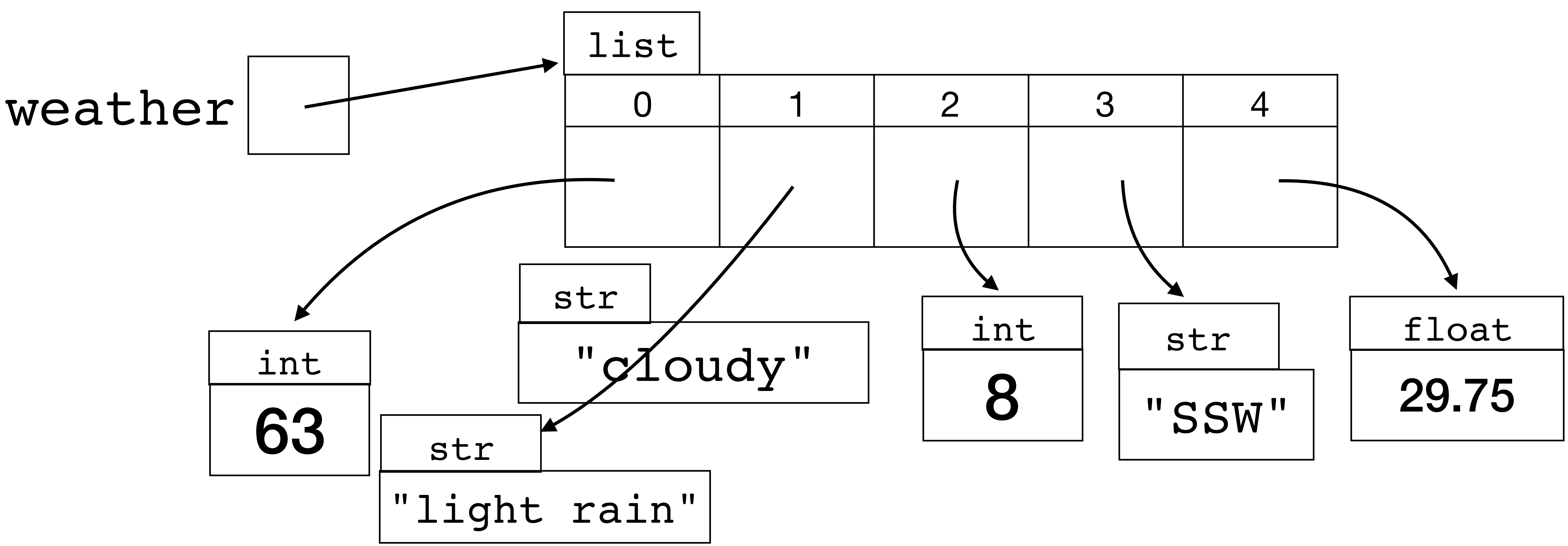
List elements store references to objects

```
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```



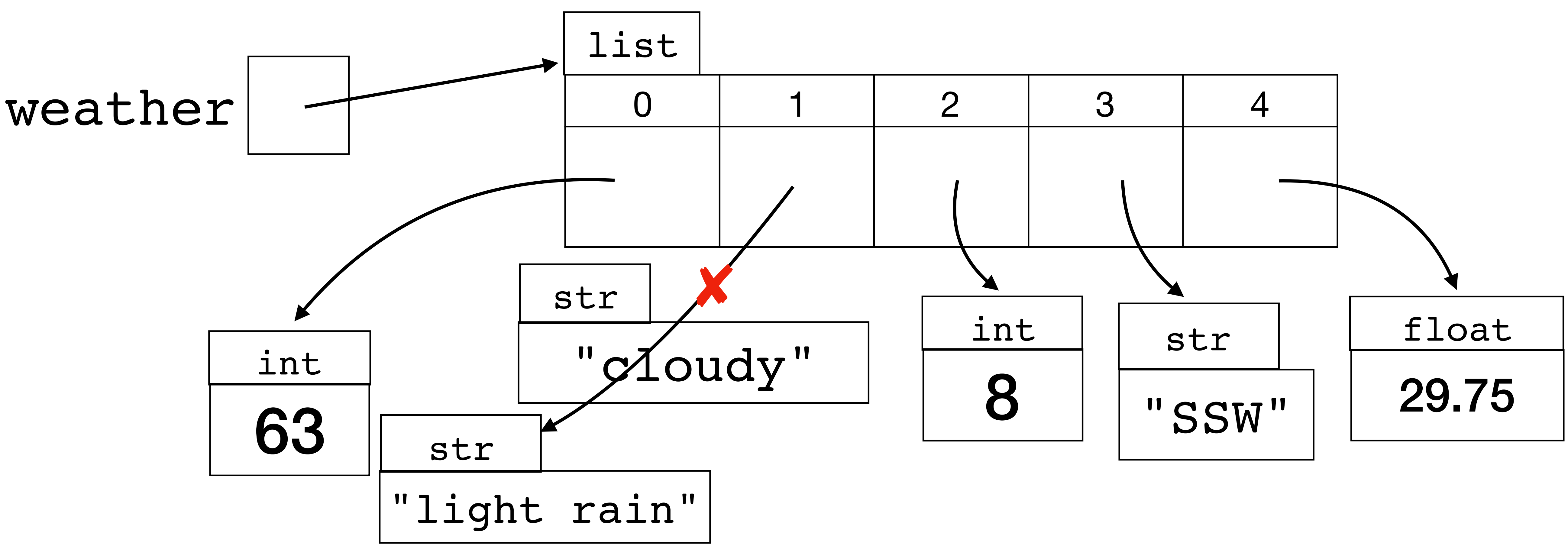
List elements store references to objects

```
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```



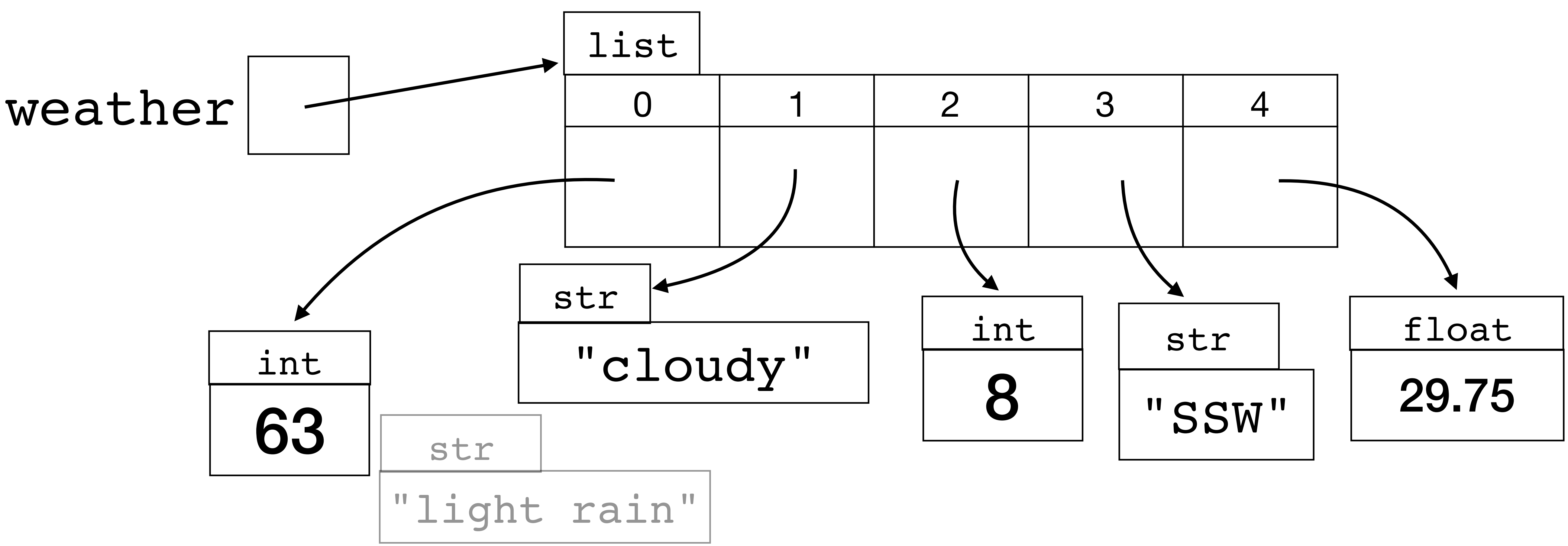
List elements store references to objects

```
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```



List elements store references to objects

```
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```



Implications of Mutability

```
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```

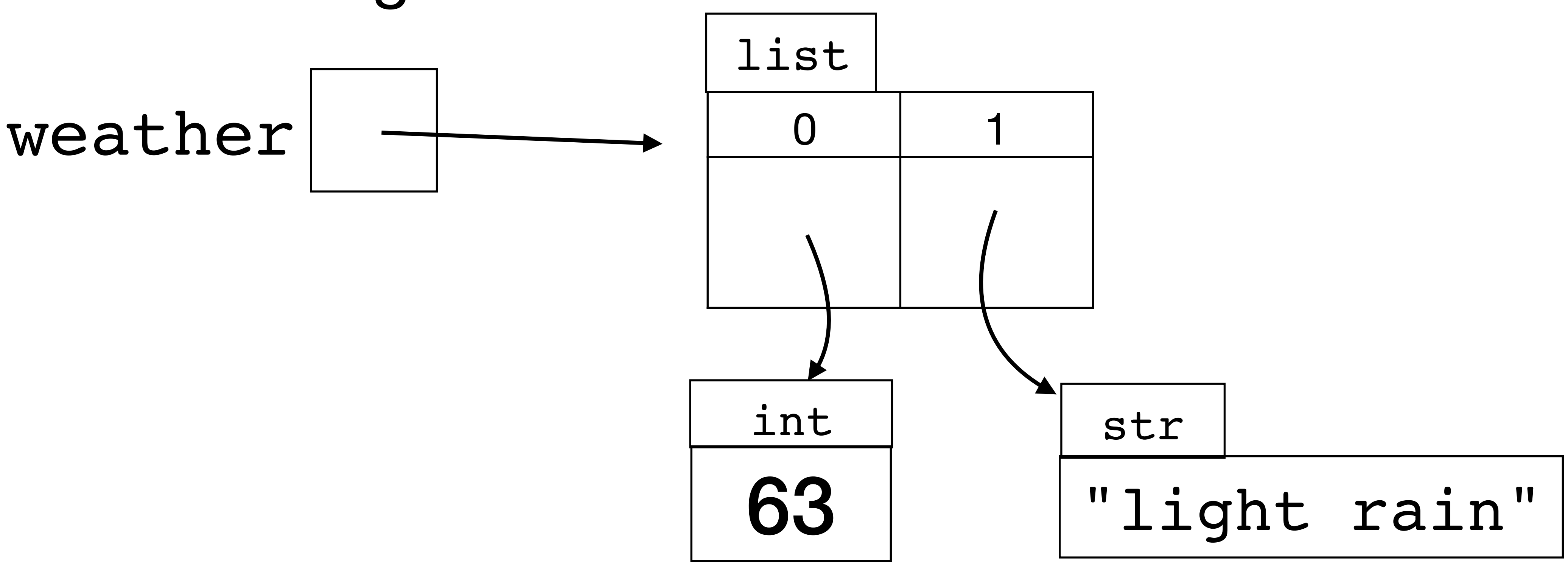
ABCD: What does the above code print?

```
A. light rain
B. Error
C. 63
D. 68
```

Implications of Mutability

```
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```

After creating the initial list:



On the board: how does this picture change as the code is executed?

Creating lists vs Creating references

- A list literal creates a new list

```
a = [4, 5, 6]
```

- List assignment does not create a new list

```
b = a
```

- List concatenation creates a new list

```
c = a + b
```

- List slicing creates a new list

```
d = a[:1]
```

A few more list operations:

A few more list operations:

`my_list.index(value)`
Return the index of the first occurrence of `value` in `my_list`
Throw an error if `value` is not in `my_list`.

A few more list operations:

`my_list.index(value)`
Return the index of the first occurrence of `value` in `my_list`
Throw an error if `value` is not in `my_list`.

`my_list.insert(index, value)`
Inserts `value` into `my_list` at `index`, shifting all following elements one spot to the right.

A few more list operations:

`my_list.index(value)`
Return the index of the first occurrence of `value` in `my_list`
Throw an error if `value` is not in `my_list`.

`my_list.insert(index, value)`
Inserts `value` into `my_list` at `index`, shifting all following elements one spot to the right.

`my_list.remove(value)`
Removes the first item from the list whose value is equal to `value`.
Causes an error if `value` is not in `my_list`.

A few more list operations:

`my_list.index(value)`
Return the index of the first occurrence of `value` in `my_list`
Throw an error if `value` is not in `my_list`.

`my_list.insert(index, value)`
Inserts `value` into `my_list` at `index`, shifting all following elements one spot to the right.

`my_list.remove(value)`
Removes the first item from the list whose value is equal to `value`.
Causes an error if `value` is not in `my_list`.

`del my_list[index]`
Removes the element at `index`, shifting all following elements one spot to the left.

index, insert, remove, del: Demo

```
abc = ["B", "C"]
abc.index("C")
abc.index("F")
abc.insert(0, "A")
abc.remove("C")
abc.remove("F")
del abc[0]
```

Worksheet - Problem 2

Execute the following, drawing and updating the memory diagram for each variable and object involved.

```
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
```

Worksheet - Problem 2

```
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
print(a)
```

What does this print?

Worksheet - Problem 2

```
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
print(a)
```

What does this print?

A. [1, 4]
B. [4, 4]
C. [[1], [4]]
D. [[4], [4]]

Problem 3

Write a function that returns a true copy (i.e., a different list object containing the same values) of a given list.

```
def copy_list(in_list):
    """ Return a new list object containing
        the same elements as in_list.
        Precondition: in_list's contents are
        all immutable. """
```

Problem 3

Write a function that returns a true copy (i.e., a different list object containing the same values) of a given list.

```
def copy_list(in_list):
    """ Return a new list object containing
        the same elements as in_list.
        Precondition: in_list's contents are
        all immutable. """
```

Hint: one possible approach uses a loop and the `append` method.

Problem 4

```
def snap(avengers):
    """ Remove a randomly chosen half of the
        elements from the given list of avengers
    """
```