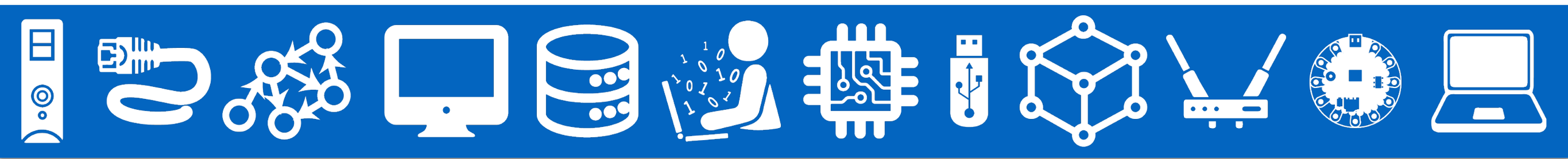
CS 134:
Mutability



Announcements & Logistics

- **HW 5** will be released today
- **Lab 4**
  - Part 1 due today/tomorrow
    - Must hand-in Part 1 to get credit for Part 2!
    - No Lab Extensions on Part 1
  - Part 2 due *next* Wednesday/Thursday
  - runtests.py: `least_votes_test4()` should return `['Diz', 'Mo']`, **not** `['Mo']`. So, change the *should return* print line to:
    - `print(" should return: ['Diz', 'Mo']")`
- **Midterm Exam** is Thursday, October 17 at 6pm or 8pm
- **Final Exam** schedule is posted: Wednesday, December 11 at 9:30am

Do You Have Any Questions?

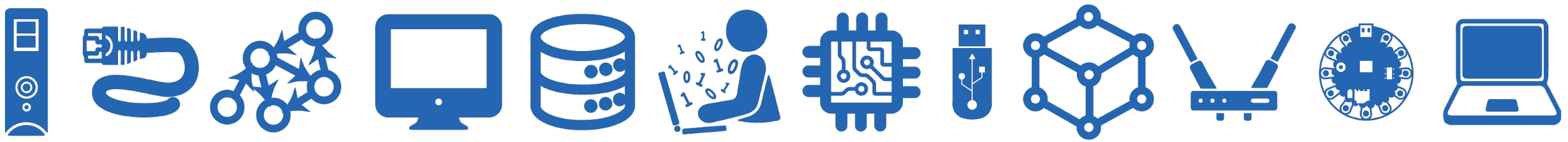
Last Time

- **Querying** sequences of sequences
  - Finding the max/min counts, writing **helper functions**
- **Module vs scripts**
  - How to import and test functions
  - Role of the special `if __name__ == "__main__":` code block
    - Great for testing code in Lab Assignments - so long as we didn't already provide an `if __name__` block!

Today's Plan

- Discuss **mutability** and its consequences: **list aliasing**

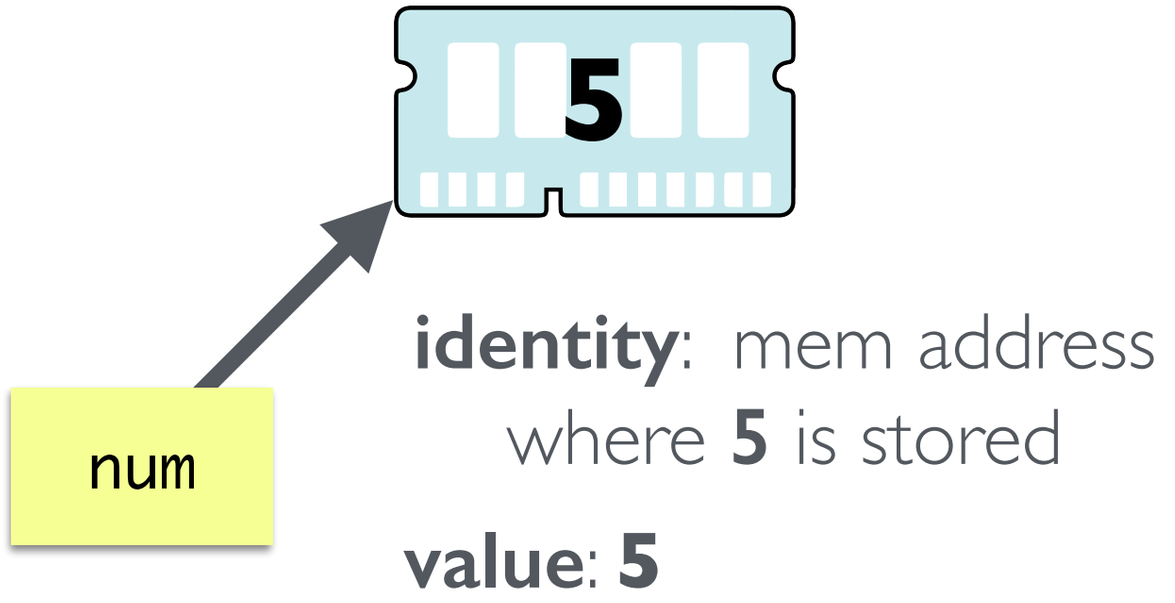
Mutability, Identity, and Value



Value vs Identity

- Python is an **object oriented language:** everything is an object!
- An **object's identity** never changes once it has been created; think of it as the object's ***address*** in memory
  - The `id()` function returns an integer representing an object's identity (or address)
- An **object's value** is the value assigned to the object when it is created

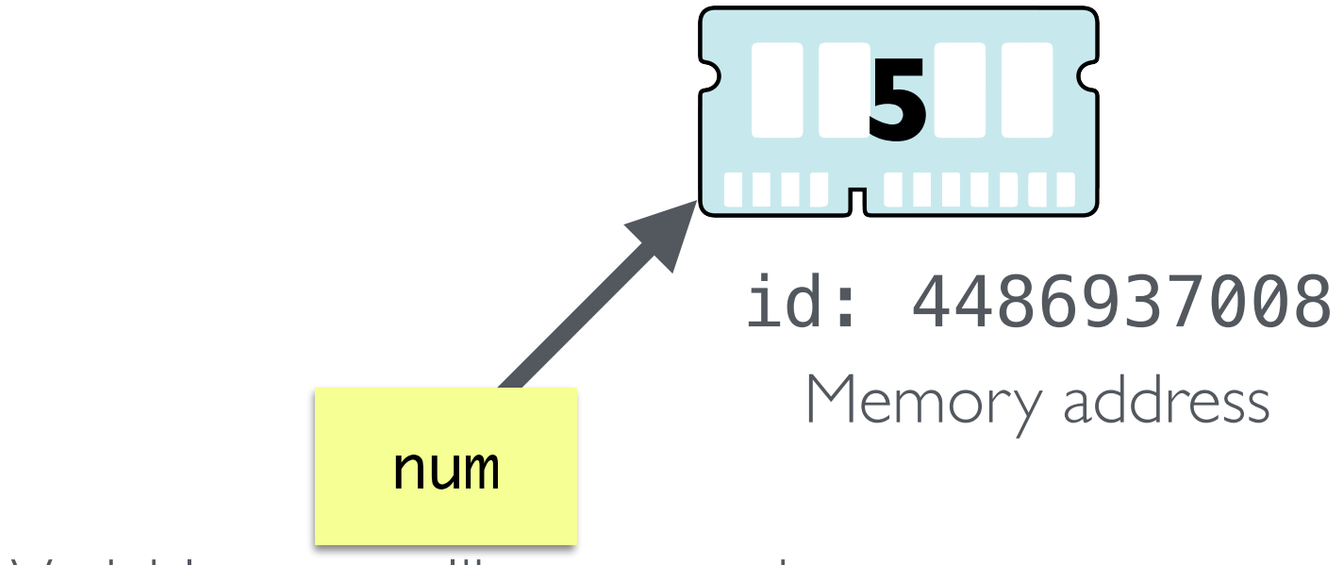
```
>>> num = 5
>>> id(num)
4486937008
```



Value vs Identity

- An **object's identity** never changes once it has been created; think of it as the object's ***address*** in memory
- On the other hand, an **object's value** can change
- Objects whose values can change are called **mutable**; objects whose values cannot change are called **immutable**

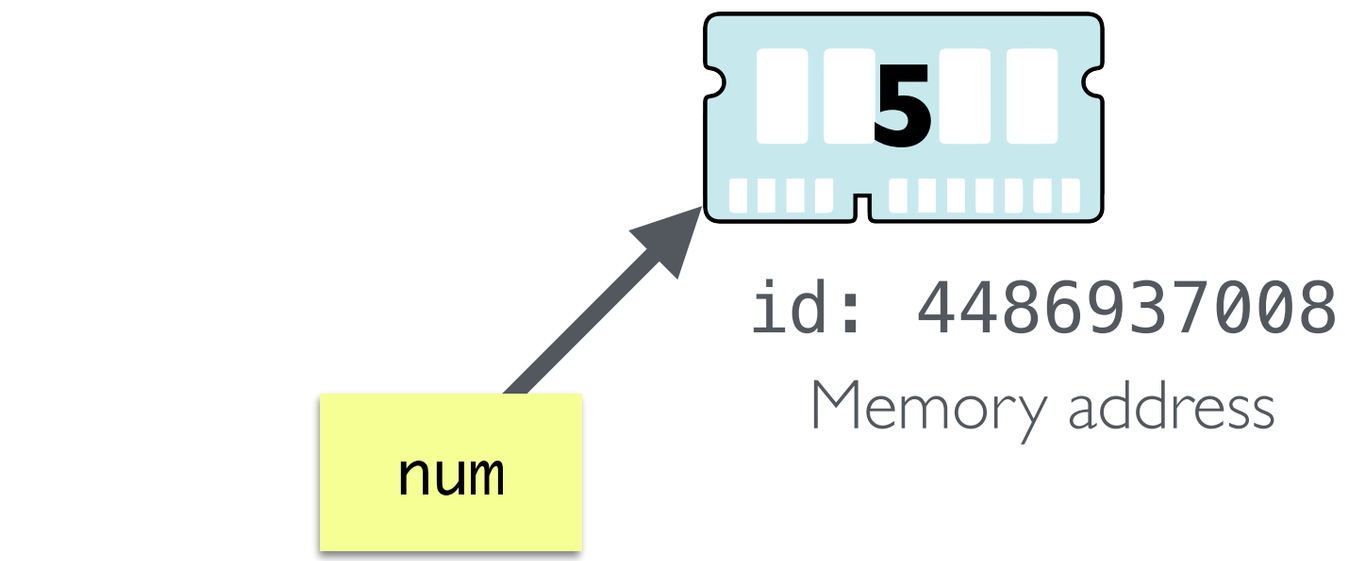
```
>>> num = 5
>>> id(num)
4486937008
```



Comparing Value vs Identity

- The `==` operator compares the **value** of an object (i.e., are the contents of the objects the same?)
- The `is` operator compares the **identity** of two objects (i.e., do they have the same memory address?)
    - `var1 is var2` is equivalent to `id(var1) == id(var2)`

```
>>> num = 5
>>> id(num)
4486937008
```



Mutability in Python

Strings, Ints, Floats are Immutable

- Once you create them, their value **cannot** be changed!
- All functions and methods that manipulate these objects return a ***new object*** and ***do not modify*** the original object

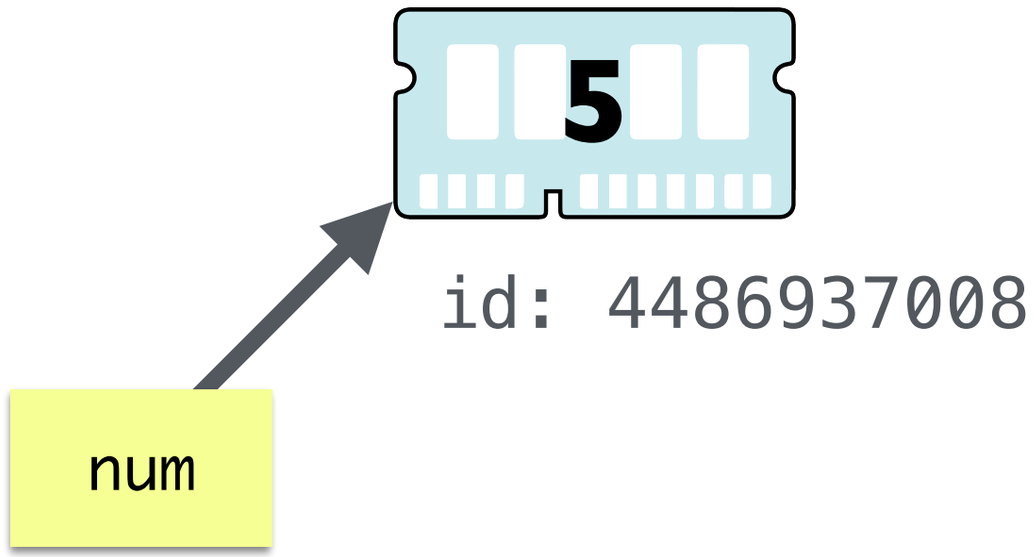
Lists are Mutable

- List values **can** be changed, e.g., by using the indexing notation and directly modifying an element of the list. There are also exist list methods that can be used to modify the list in place
- If we use sequence operators on lists, these functions and operations return a ***new list*** and ***do not modify*** the original list

Ints, Floats are Immutable

```
>>> num = 5
>>> id(num)
4486937008

>>> num = num + 1
>>> id(num)
```

Has the identity of **num** changed?

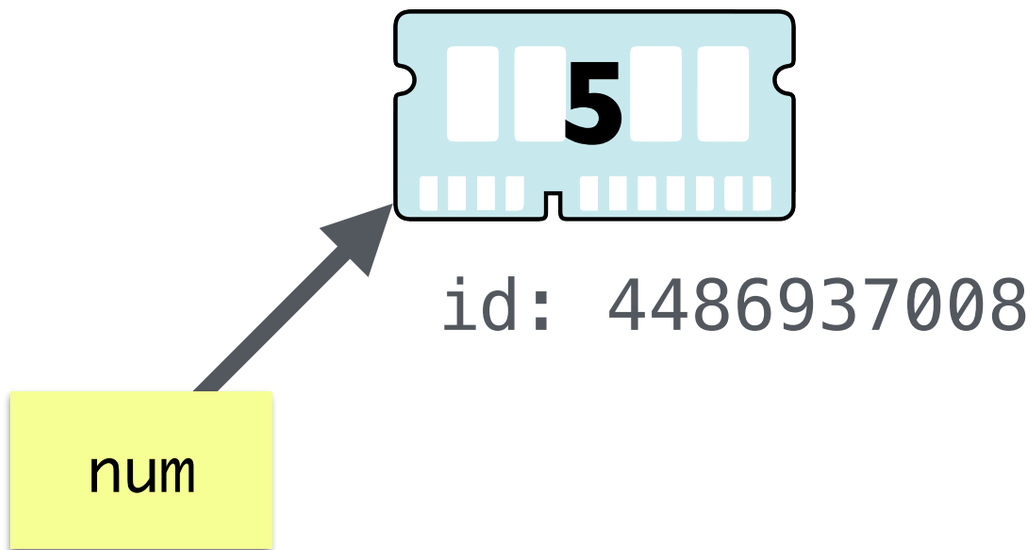


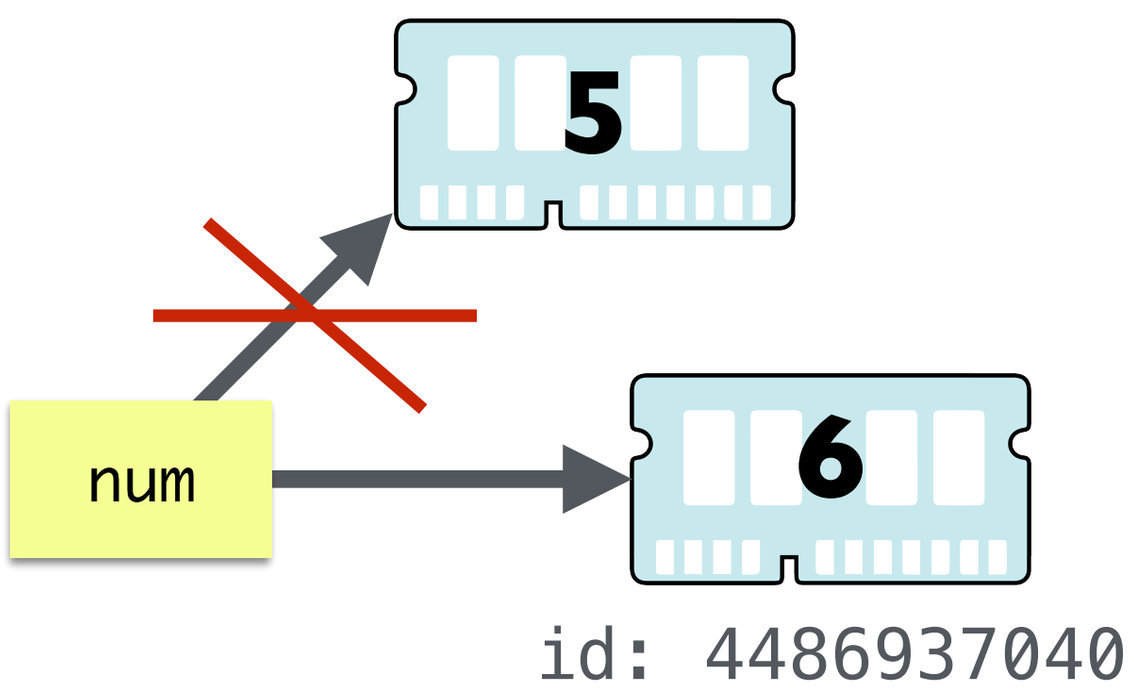
Attempts to change an immutable object create a new object

Ints, Floats are Immutable

```
>>> num = 5
>>> id(num)
4486937008

>>> num = num + 1
>>> id(num)
4486937040
```

Identity of ints cannot be changed, `num` assumes a **new** identity

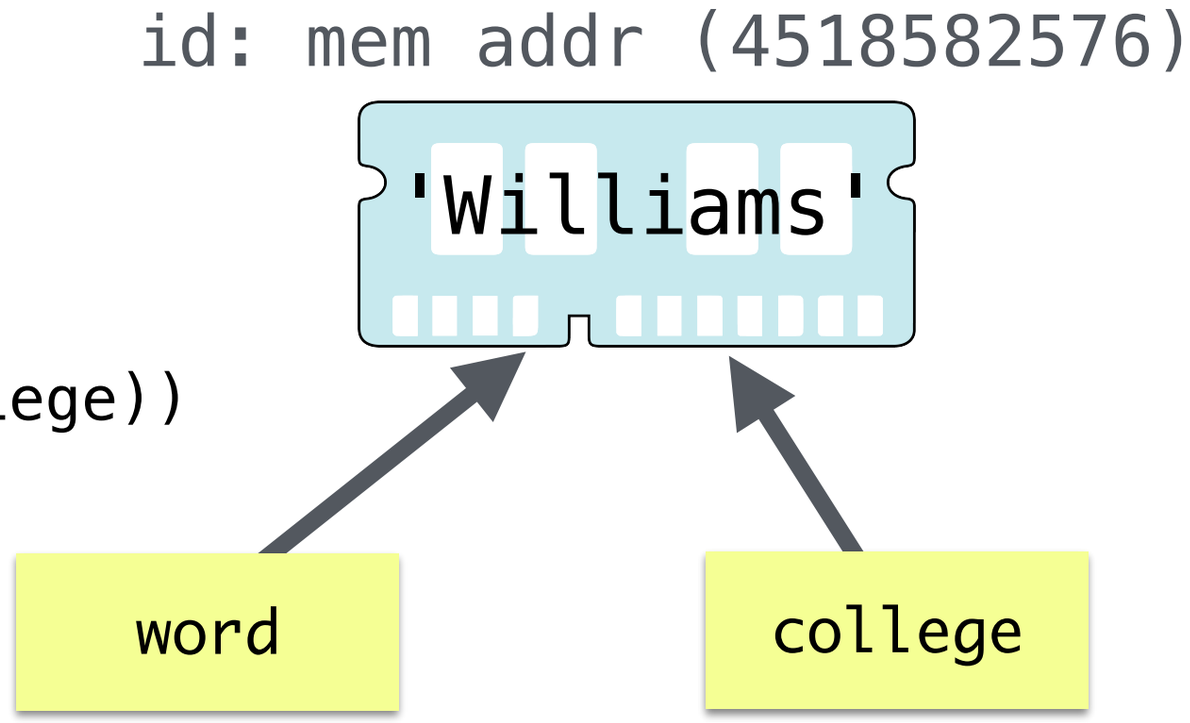




Attempts to change an immutable object create a new object

Strings are Immutable

```
>>> word = "Williams"
>>> college = word
>>> word == college
True

>>> print(id(word), id(college))
4518582576 4518582576

>>> word is college
True
```



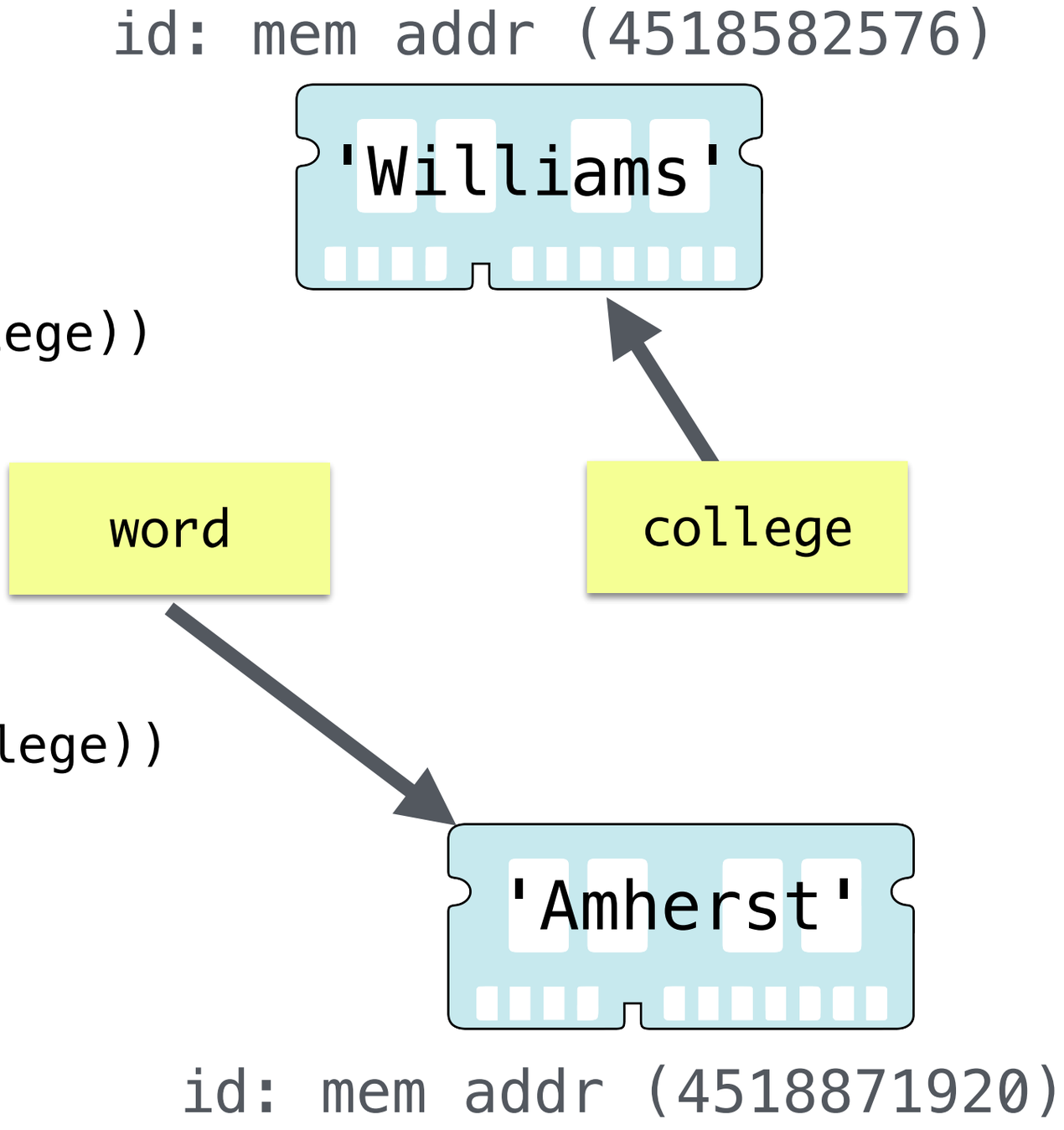
Variable names point to memory addresses of stored value

Even though word and college have the same identity and value, if we update one of them, it just assumes a new identity!

Attempts to change an immutable object create a new object

Strings are Immutable

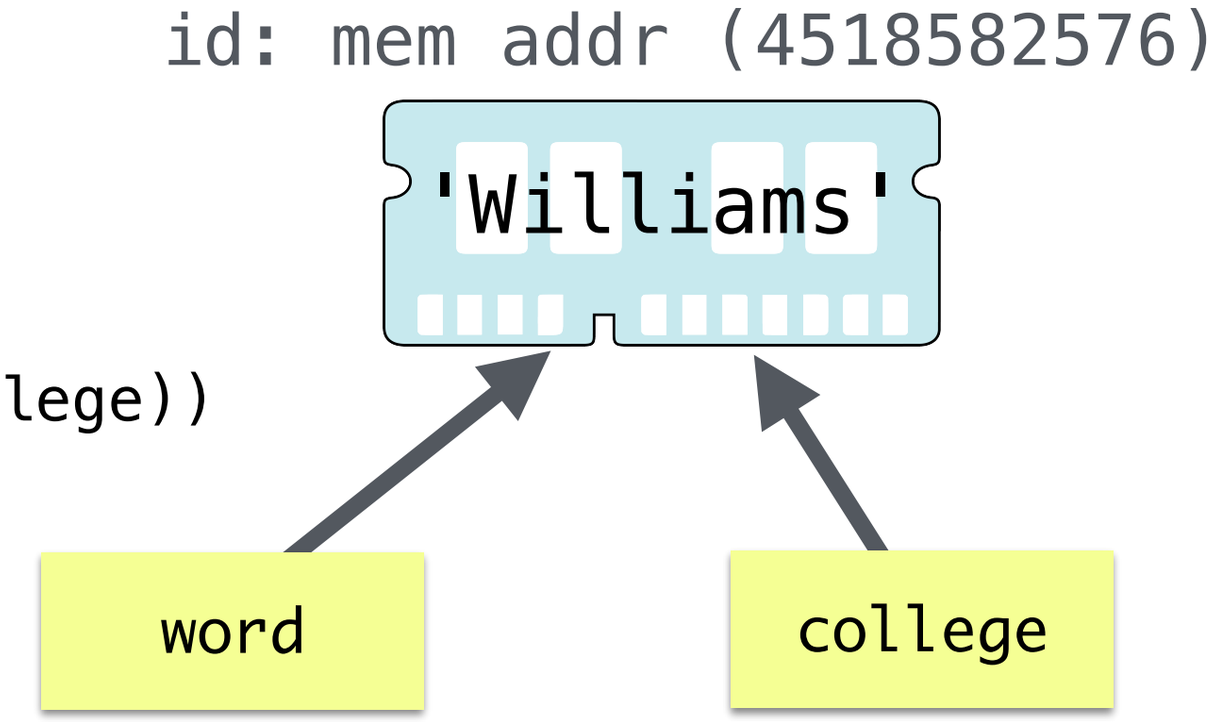
```
>>> word = "Williams"
>>> college = word
>>> word == college
True

>>> print(id(word), id(college))
4518582576 4518582576

>>> word is college
True

>>> word = "Amherst"
>>> print(id(word), id(college))
4518871920 4518582576

>>> word is college
False
```



Attempts to change an immutable object create a new object

Strings are Immutable

```
>>> word = "Williams"
>>> college = "Williams"
>>> word == college
True

>>> print(id(word), id(college))
4518582576 4518582576

>>> word is college
True
```



Variable names point to memory addresses of stored value

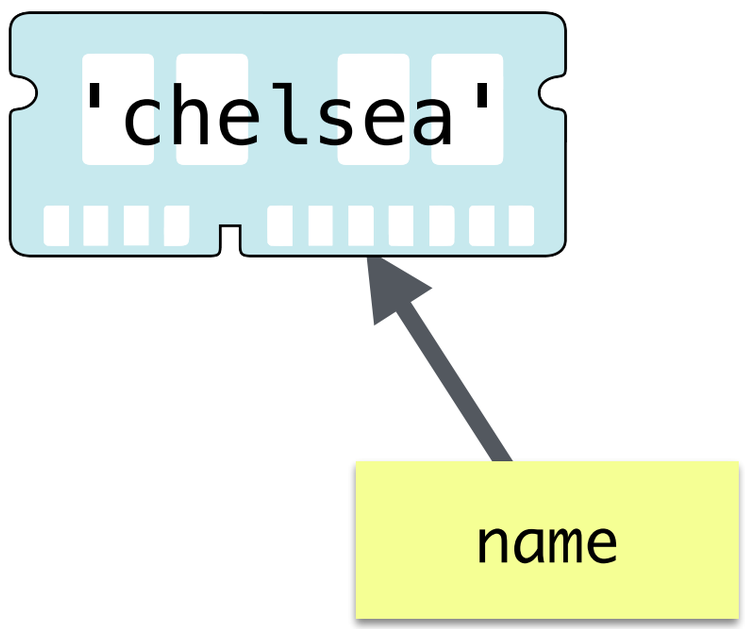
Even though we created word and college separately, they still point to the same memory address. This is a (confusing) optimization in Python.

Immutable objects that are == also share an identity

String Operations Return New Strings

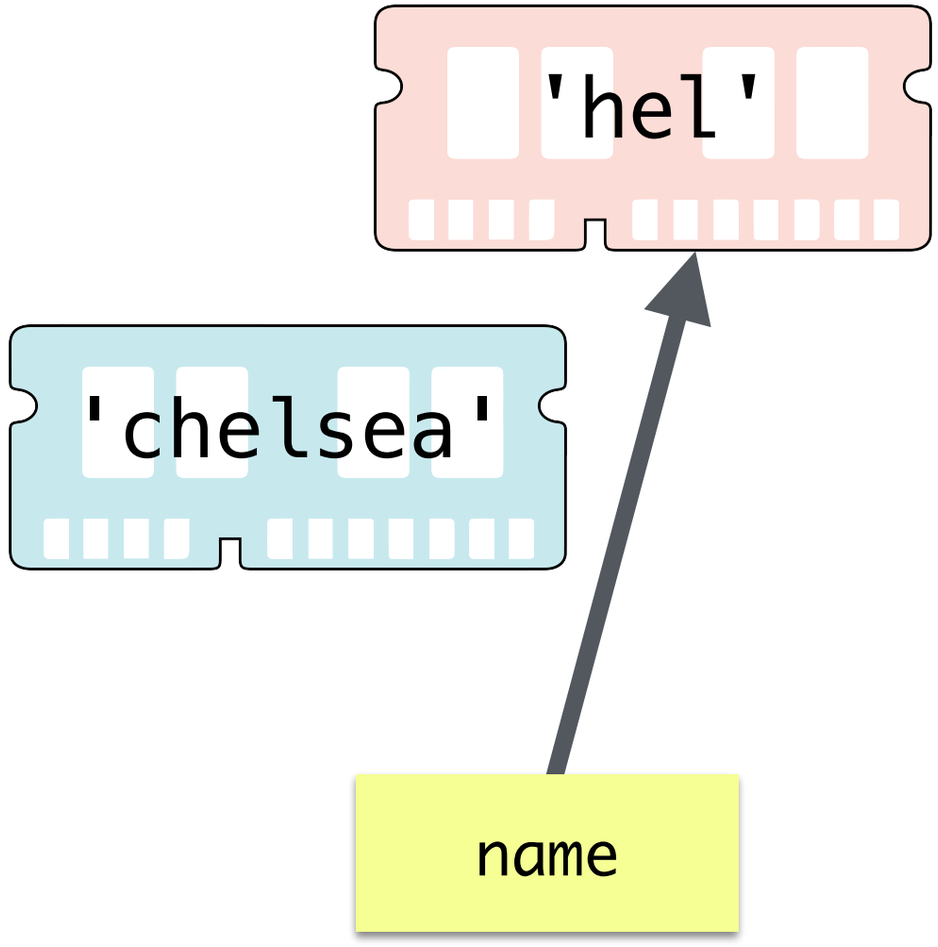
- Sequence operations, like slicing `[:]`, return **new sequences**

```
>>> name = "chelsea"
>>> id(name)
4574657776
```



String Operations Return New Strings

- Sequence operations, like slicing **`[:]`**, return **new sequences**

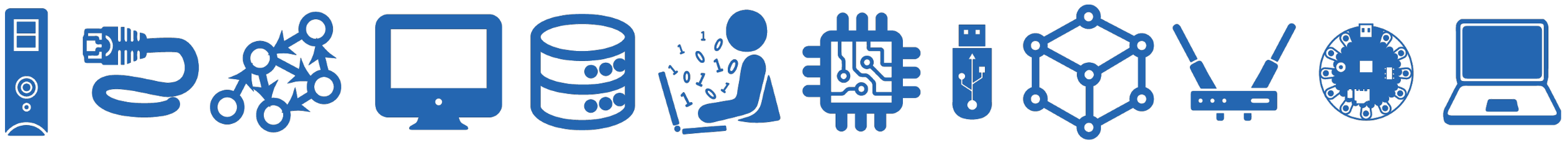



```
>>> name = "chelsea"
>>> id(name)
4574657776

>>> name = name[1:4]
>>> id(name)
4574684720
```

Sequence Operations Return New Sequences

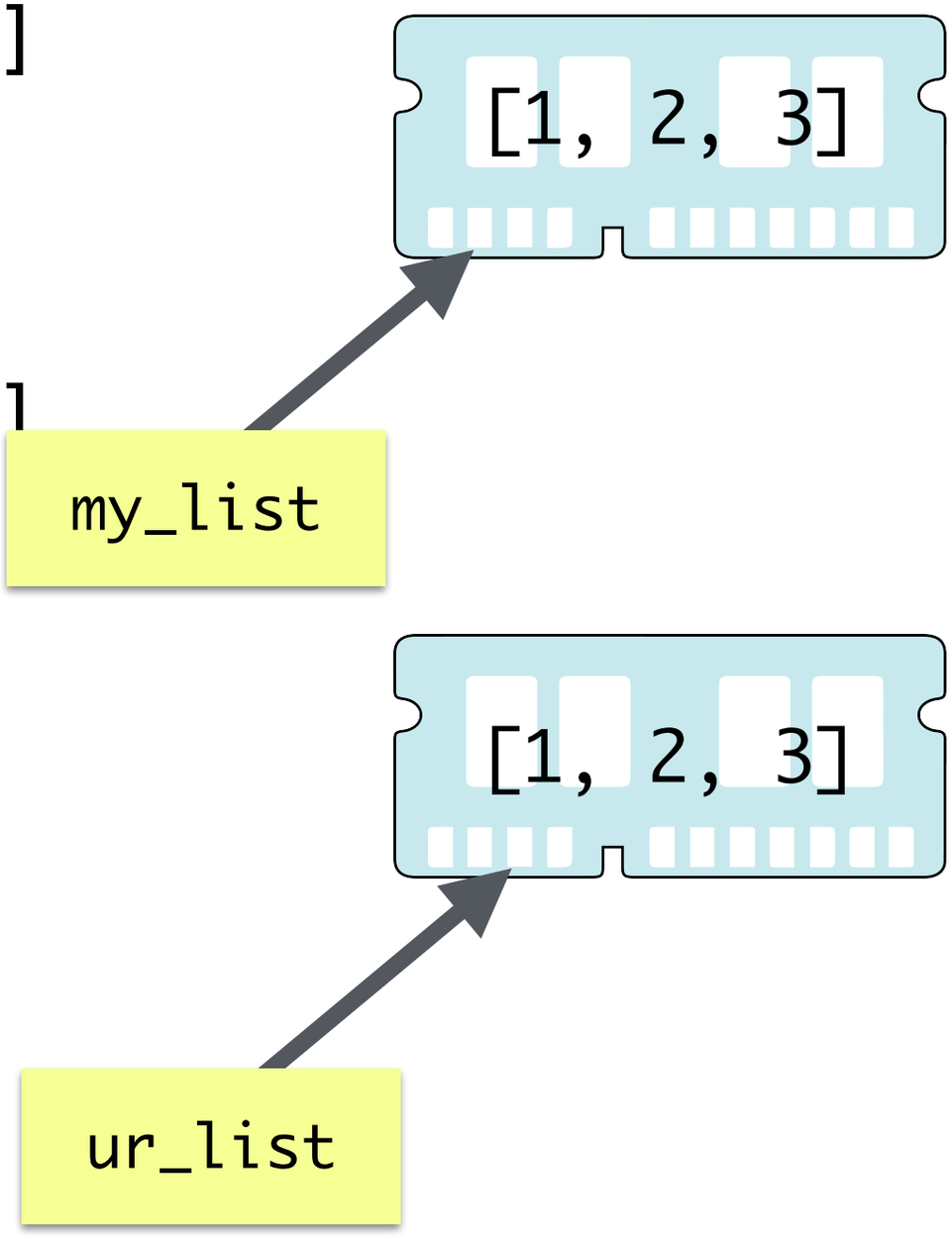
- The following operations, that can be performed on both **lists** and **strings**, and always return a **new list/string**
  - `[::]` slicing operator: returns a new sliced sequence
  - assignment of a new sequence to a variable
    - `names = 'Iris and Lida'`
    - `my_list = [1, 2, 3]`
- concatenation (**+**) always creates a new sequence

List
Identity and Value



Lists are Mutable

```
>>> my_list = [1, 2, 3]
>>> id(my_list)
4418551104

>>> ur_list = [1, 2, 3]
>>> id(my_list)
4418559392
```



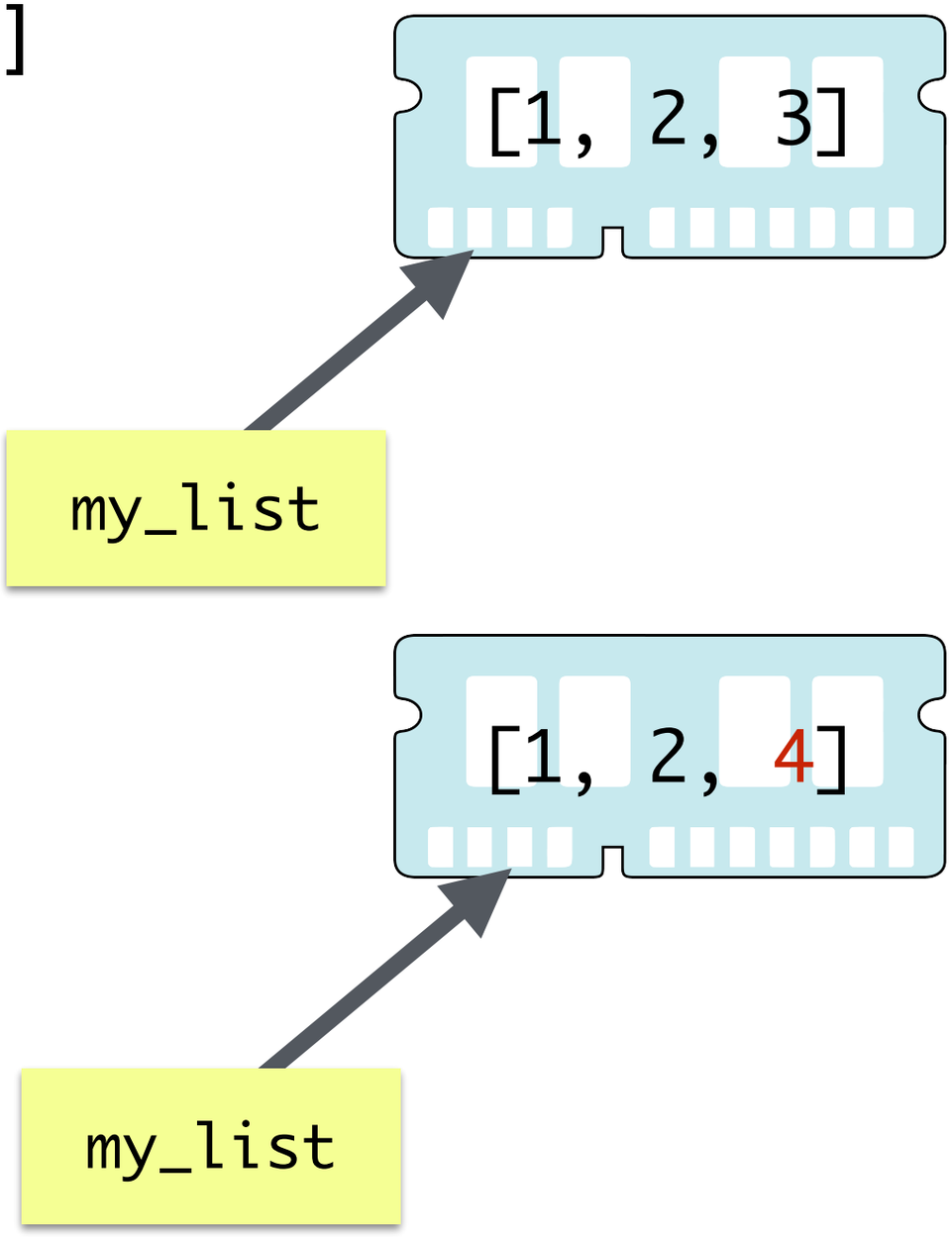
Note: Same value, different identities

List objects created separately with same values, different identities

Lists are Mutable

```
>>> my_list = [1, 2, 3]
>>> id(my_list)
4418551104

>>> my_list[-1] = 4
>>> id(my_list)
4418551104
```

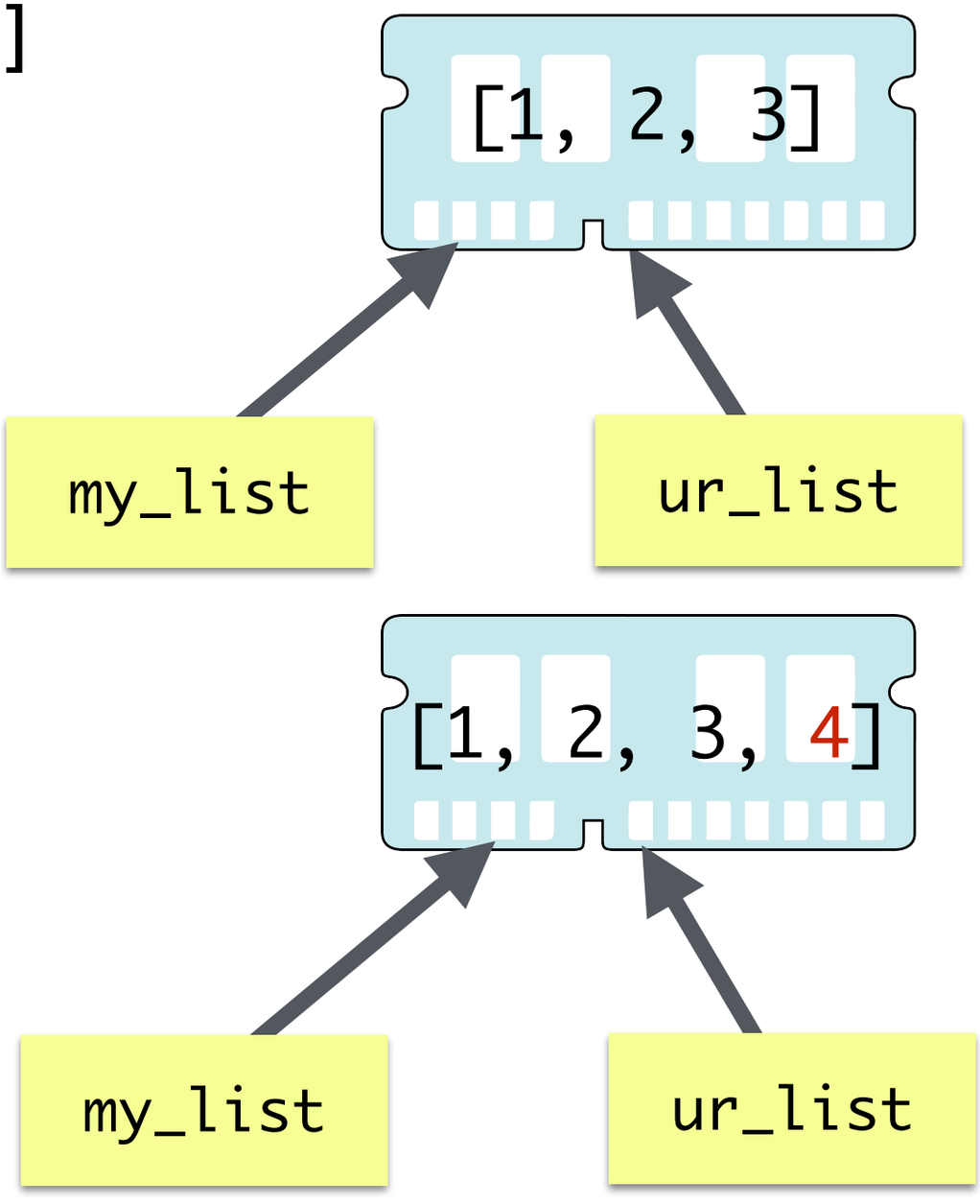
Note: Value changes, identity stays the same



Value of list objects can change, keeping identity the same

Lists are Mutable

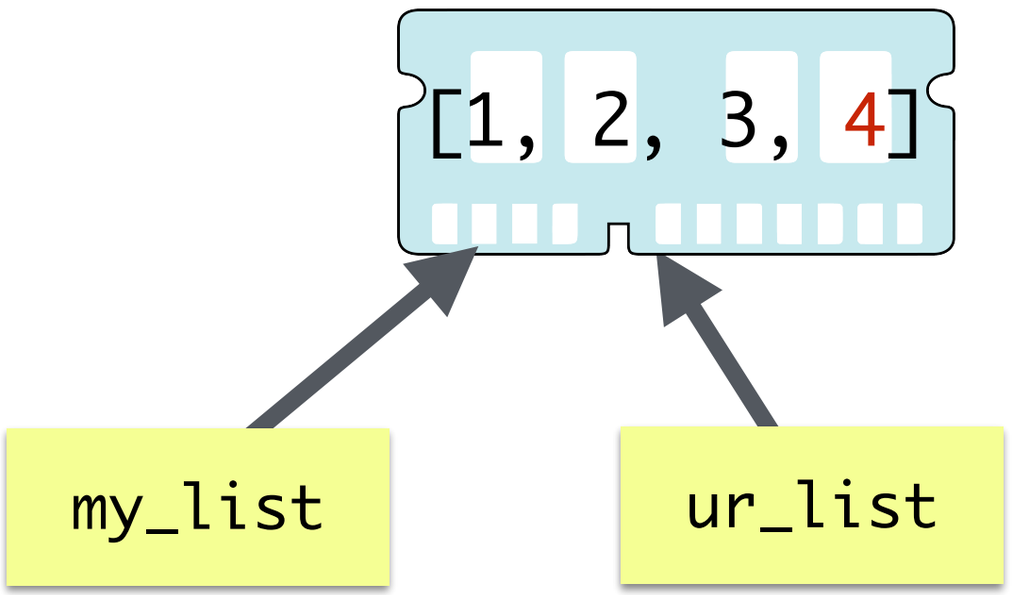
```
>>> my_list = [1, 2, 3]
>>> id(my_list)
4418551104

>>> ur_list = my_list
>>> id(ur_list)
4418551104

>>> my_list += [4]
>>> ur_list
[1, 2, 3, 4]
```



Two variables, one identity, changes to one impact the other!

List Appending vs. Concatenation

```
>>> my_list += [4]
>>> ur_list
[1, 2, 3, 4]
```




```
>>> my_list = my_list + [5]
>>> ur_list
[1, 2, 3, 4]
```

+= (appending) modifies a list, but + (concatenation) does not!

Mutability in Python

Strings, Ints, Floats are Immutable

- Once you create them, their value **cannot** be changed!
- All functions and methods that manipulate these objects return a ***new object*** and ***do not modify*** the original object

Lists are Mutable

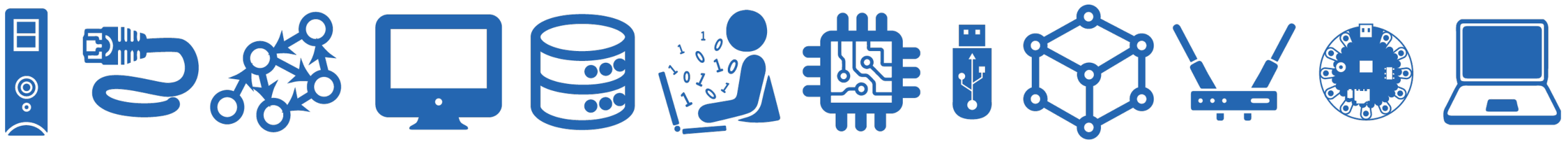
- List values **can** be changed
- Sequence operators and functions return a ***new list; do not modify*** the original list
- List methods ***modify*** what's in a list
- The **mutability** of lists has many implications such as ***aliasing***
- **Aliasing** happens when the value of one variable is assigned to another variable
  - Can have multiple names for the same object!

Mutability: Take-Aways

- Everything in Python is an **object** and has a memory address
- When we check to see if two objects `is` each other we're checking the memory address of the objects
- When we check to see if two objects `==` each other, we're checking the values

- `list` objects are **mutable** or *changeable*. We can change them by:
  - ...placing an indexed list on the lefthand side of an assignment operator: `my_lst[-1] = "puppy"`
  - ...or by using the append operator: `my_list += "add this item"`
- All other data structures we've seen so far are **immutable** or ***unchangeable*** --> we always end up creating a new object!

Mutability & Copying Sequences



Copying Sequences

- We can make a copy of sequences using *slicing:*

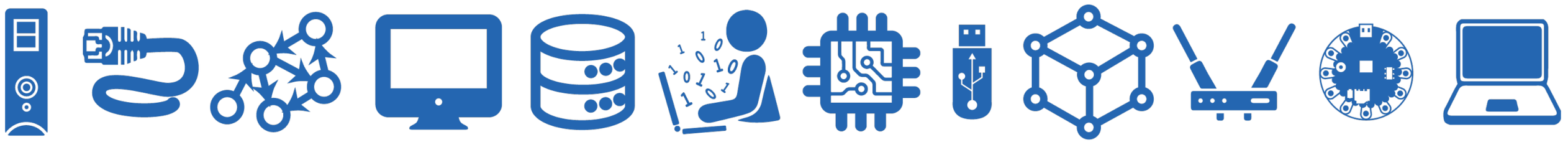
```
>>> book = ["see", "spot", "run"]
>>> book2 = book[:]
>>> book2
['see', 'spot', 'run'
```

- Or through *for..loops:*

```
>>> book3 = []
>>> for word in book:
...     book3 += word
>>> book3
['see', 'spot', 'run'
```

Why might a sequence of mutable objects require a different approach?

Mutability & Functions



Mutability and Functions

- What is the value of **`my_lst`** at the end of this code?

```
def do_something(any_lst):
    any_lst += [42]

if __name__ == "__main__":
    my_lst = [1, 2]
    do_something(my_lst)
```

Mutability and Functions

- What is the value of **`my_lst`** at the end of this code?

```
def do_something(any_lst):
    any_lst += [42]

if __name__ == "__main__":
    my_lst = [1, 2]
    do_something(my_lst)
```

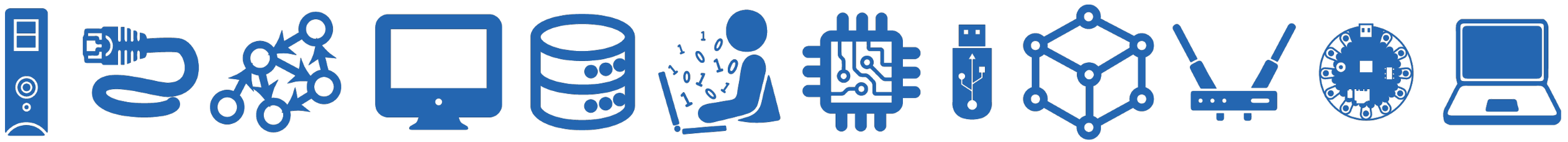
```
[1, 2, 42]
```

- Why?

When you **change** a mutable object, everything that points to it will reflect that change!

List Aliasing

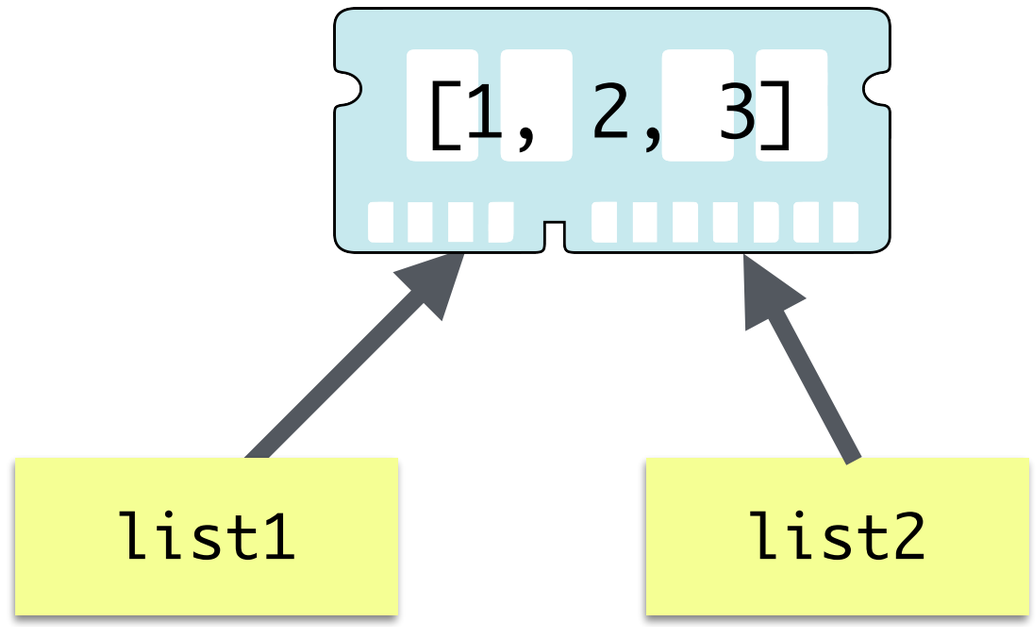
A side effect of mutability



List Aliasing

- Any assignment or operation that creates a new name for an existing object implicitly creates an ***alias*** (a new name)
- Because list objects **can change**, this leads to some unusual aliasing side effects

```
>>> list1 = [1, 2, 3]
>>> list2 = list1

>>> list1 is list2
True
```

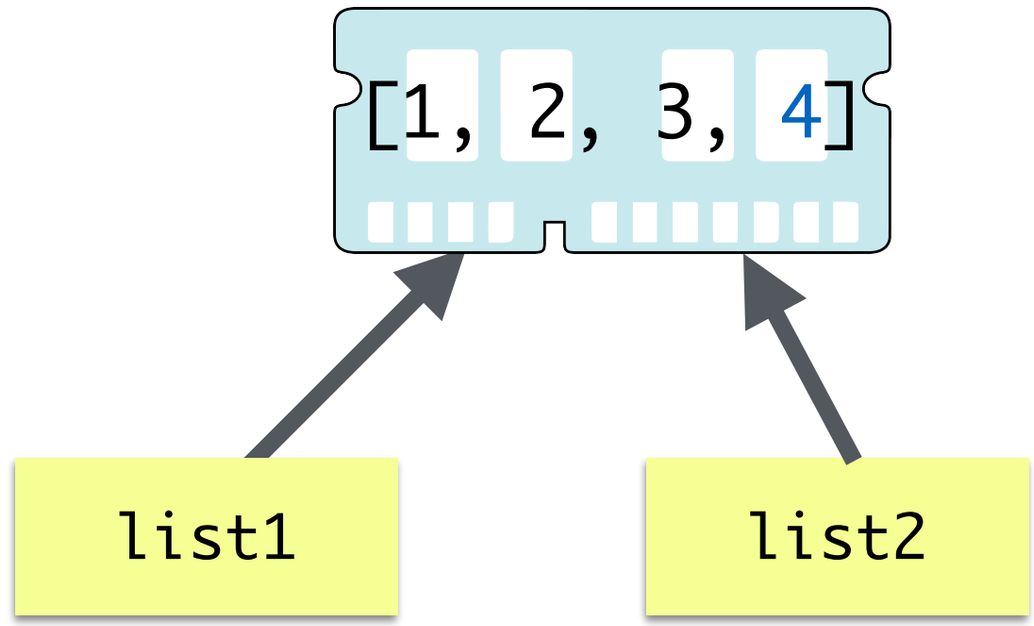


We are not creating a separate copy, but rather creating a **second name** for the original list; **`list2` is an alias of `list1`**

List Aliasing

- Unlike immutable objects (recall our string example with `word` and `college`) , changing the value of `list1` **will also change the value** of `list2`:
  - They are two names for the same list!

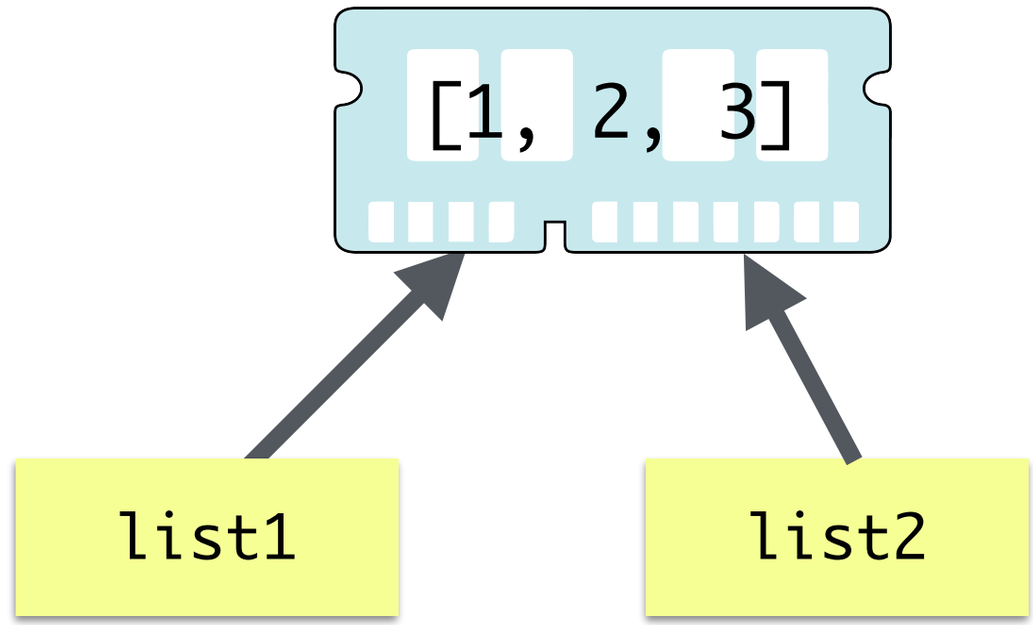
```
>>> list1 = [1, 2, 3]
>>> list2 = list1

>>> list1 is list2
True

>>> list1 += [4]
>>> list2

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

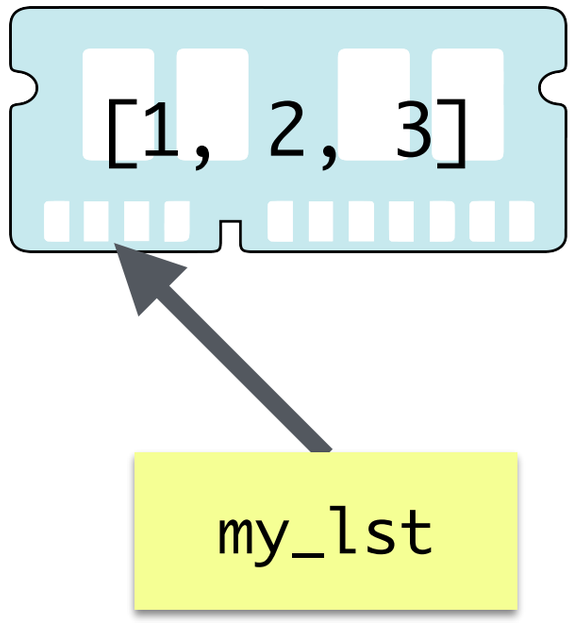


List Aliasing

- An assignment to a new variable **creates a new list**

```
>>> list1 = [1, 2, 3]
>>> list2 = list1
>>> my_lst = [1, 2, 3]

>>> # same values?
>>> my_lst == list1 == list2
True

>>> # same identities?
>>> my_lst is list1
False
```





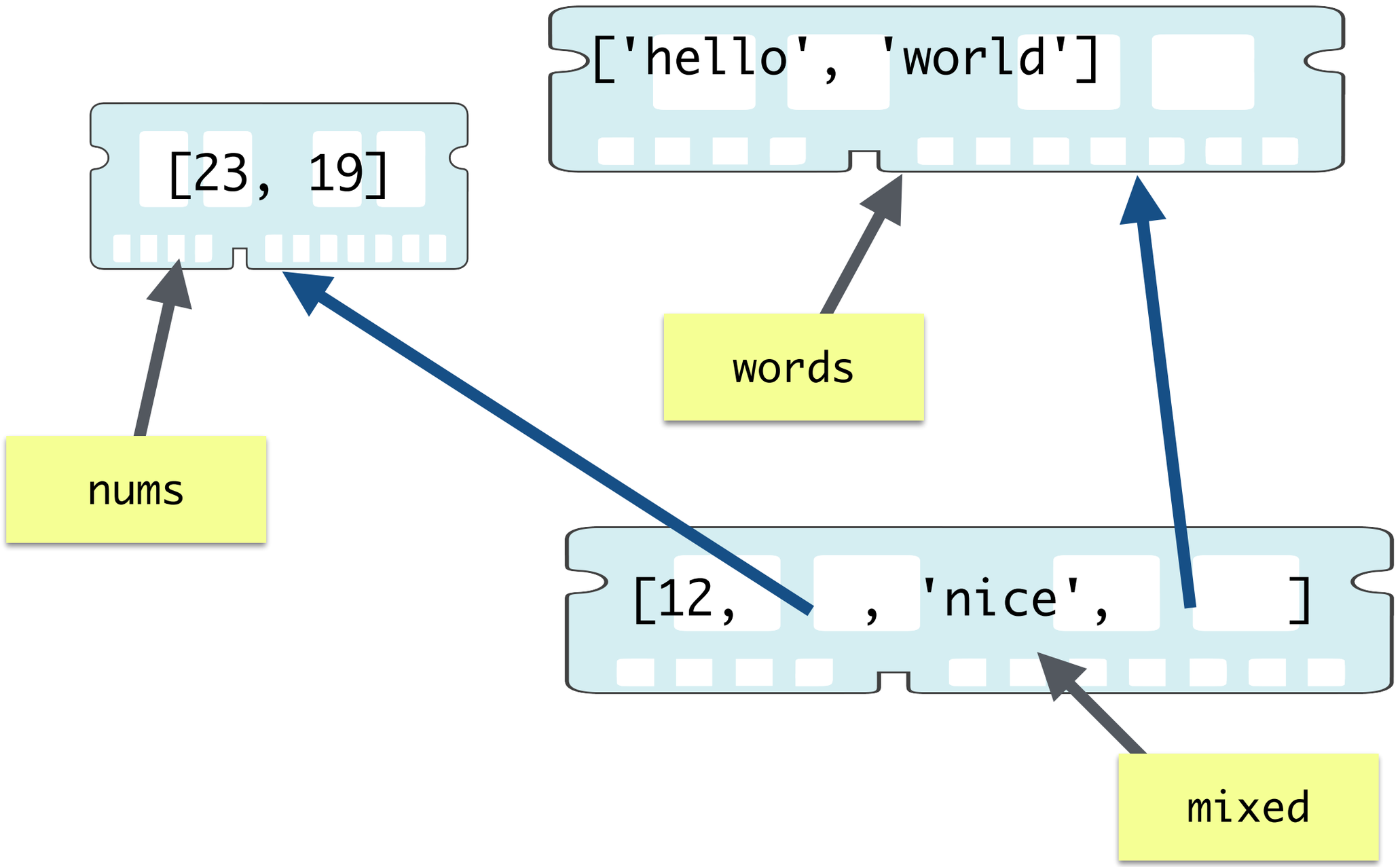
(Crazy) Aliasing Examples

```
>>> nums = [23, 19]
>>> words = ["hello", "world"]
>>> mixed = [12, nums, "nice", words]

>>> words += ["sky"]
>>> mixed
```

???

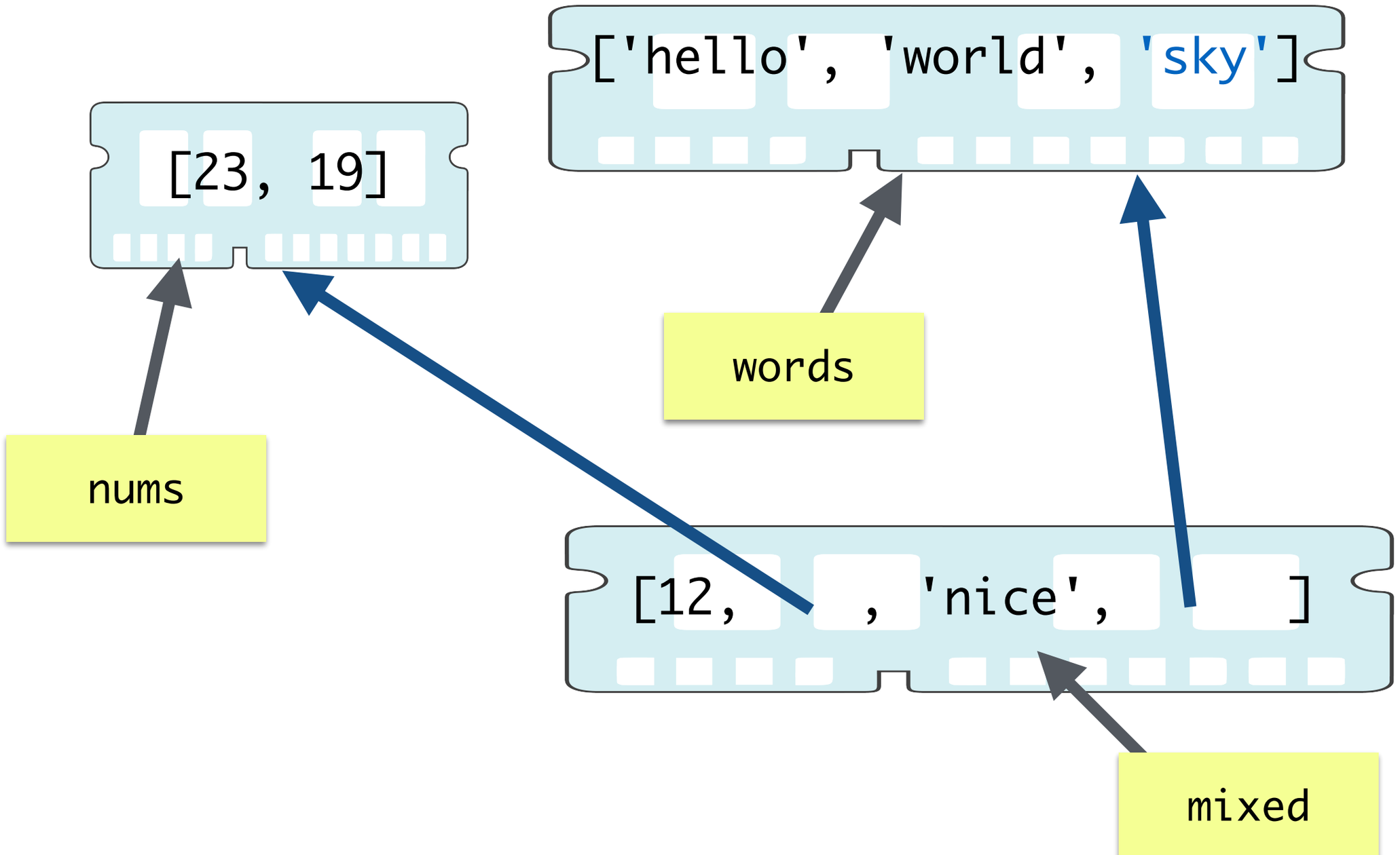
(Crazy) Aliasing Examples

```
>>> nums = [23, 19]
>>> words = ["hello", "world"]
>>> mixed = [12, nums, "nice", words]
```



(Crazy) Aliasing Examples

```
>>> words += ["sky"]
```



(Crazy) Aliasing Examples

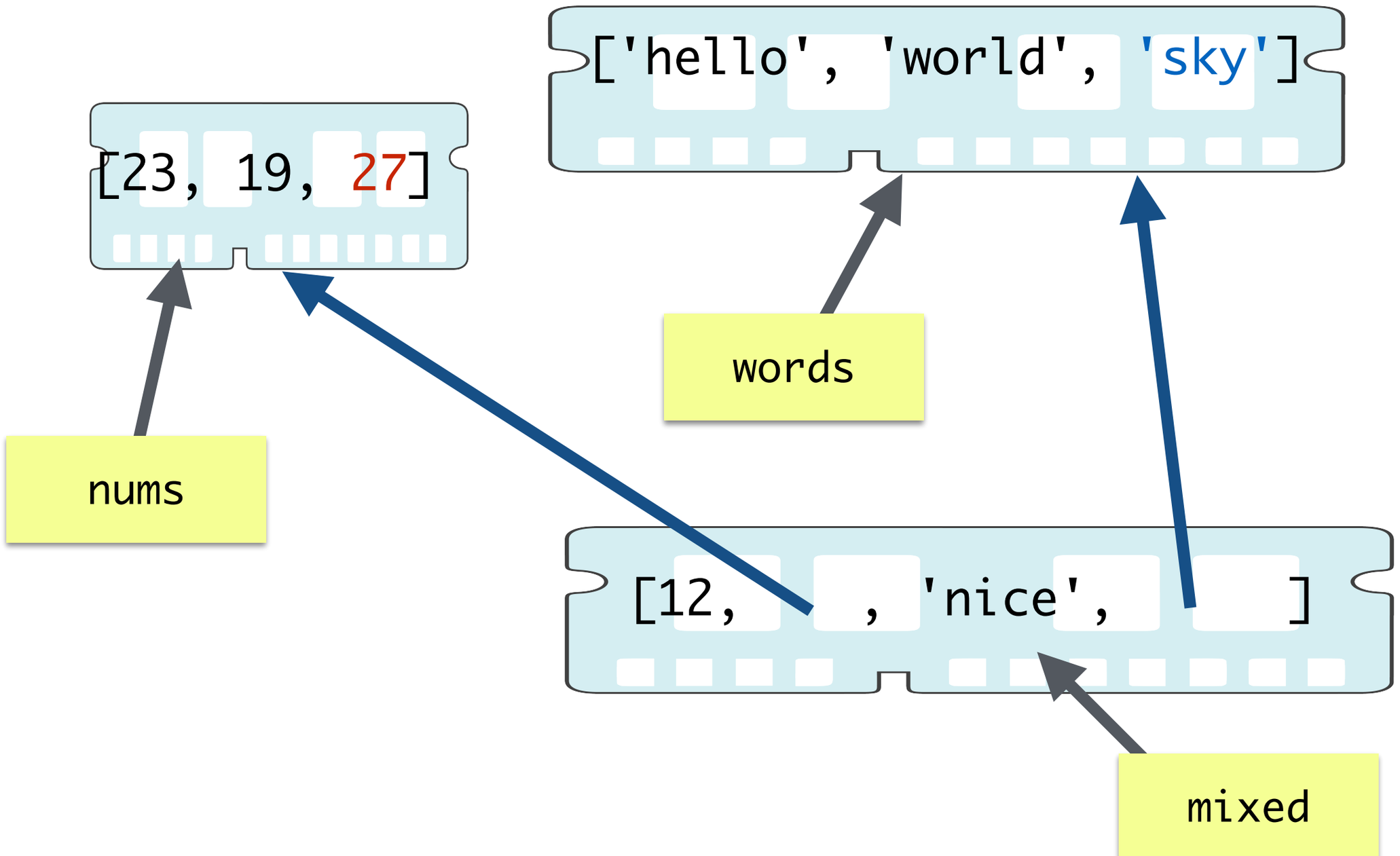
```
>>> nums = [23, 19]
>>> words = ["hello", "world"]
>>> mixed = [12, nums, "nice", words]

>>> words += ["sky"]
>>> mixed
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]
```

```
>>> mixed[1] += [27]
```

???

(Crazy) Aliasing Examples

```
>>> mixed[1] += [27]
```



(Crazy) Aliasing Examples

```
>>> nums = [23, 19]
>>> words = ["hello", "world"]
>>> mixed = [12, nums, "nice", words]

>>> words += ["sky"]
>>> mixed
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]

>>> mixed[1] += [27]
>>> nums
[23, 19, 27]
>>> mixed
[12, [23, 19, 27], 'nice', ['hello', 'world', 'sky']]
```

Conclusion

- We **cannot change** the value of **immutable** objects such as strings
  - Attempts to modify the object ALWAYS creates a new object
- We **can change** the value of **mutable** objects such as lists
  - Need to be mindful of **aliasing**; be careful to avoid unintended aliases
  - You can create a "true" copy of a list using slicing or a list
    - `new_lst = my_lst[:]`
    - `new_lst = []`

      ```
      for ele in my_lst:
          my_lst += ele
      ```
  - When using the `+=` operator with lists, it mutates the list.
    - Use `my_lst = my_lst + [element]` if you want to avoid mutation

The end!