CS 134:
Aliasing and While Loops

Announcements & Logistics

- **HW 5** due Mon at 11 pm
- **Lab 4 Part 2** due next Wed/Thur at 11 pm
    - We'll send automated feedback about Part 1 later today
- **Midterm reminder**: Thur Mar 17: 6 - 7:30 pm or 8 - 9:30 pm
- **Midterm review**: Tue, Mar 15: 7 - 8:30 pm

Do You Have Any Questions?

Last Time

- Learned about writing and appending to files (and `.format()`)
- Reviewed useful list methods:
    - All of these methods modify/mutate the list:
        - `.append()`, `.extend()`, `.insert()`, `.remove()`, `.pop()`, `.sort()`
- Started discussion on **mutability** and **aliasing** in Python

Today's Plan

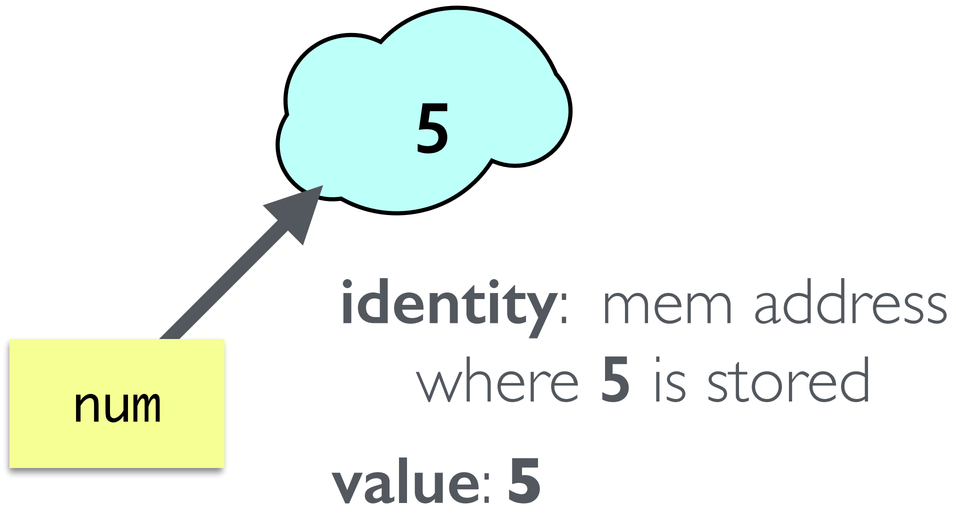
- Continue discussing **aliasing** and **mutability** in Python
- Discuss while loops
    - Needed for ranked-choice voting on Lab 4 Part 2

Mutability and Aliasing

Recap: Value vs Identity

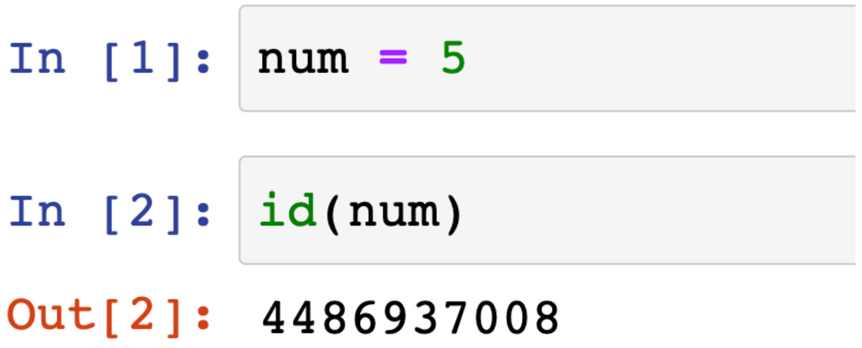
- An **object's identity** never changes in Python once it has been created; think of it as the object's ***address*** in memory
  - The `id()` function returns an integer representing an object's identity (or address)
- An **object's value** is the value assigned to the object when it is created
  - Objects whose values can change are called **mutable**; objects whose values cannot change are called **immutable**

```
In [1]: num = 5

In [2]: id(num)
Out[2]: 4486937008
```

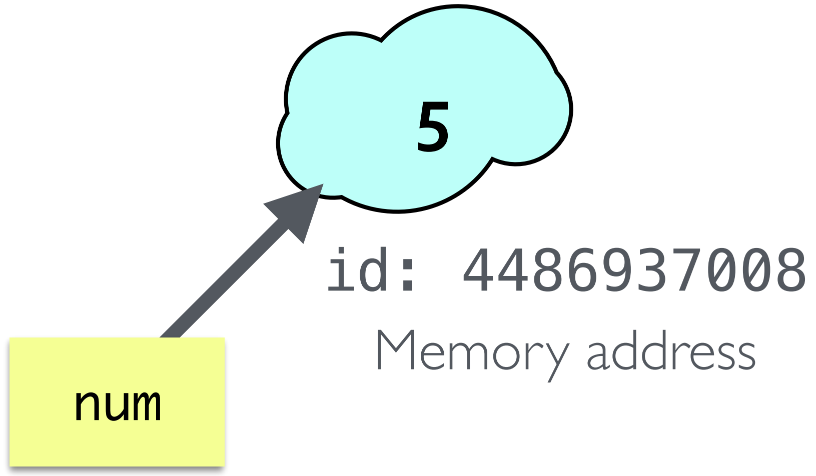


Comparing Value vs Identity

- 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)`

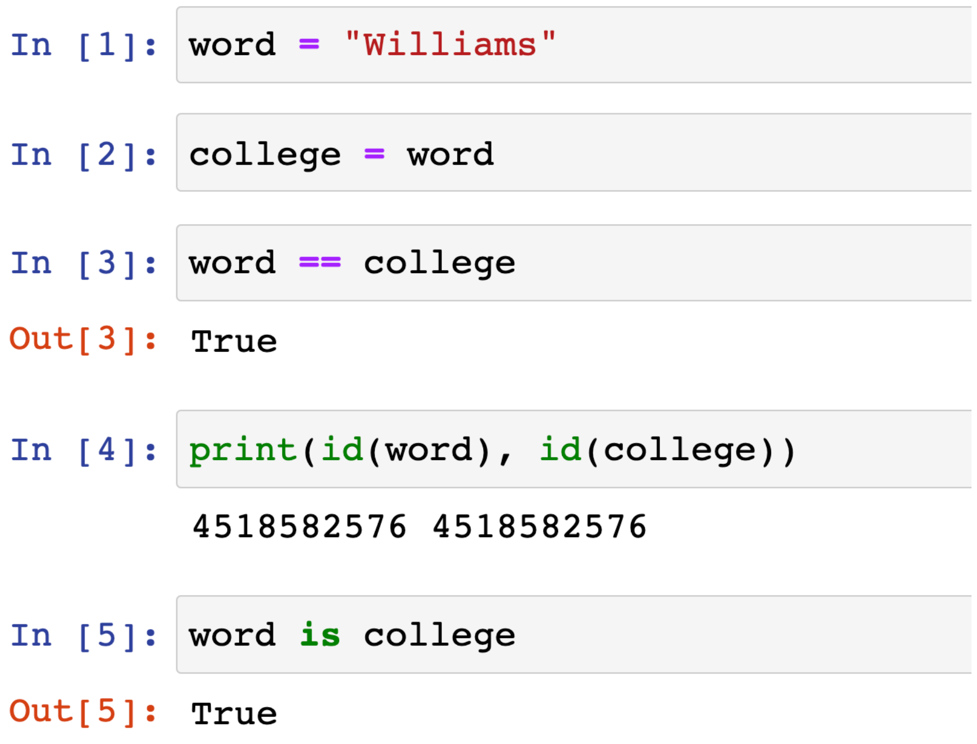


```
In [1]: num = 5

In [2]: id(num)

Out[2]: 4486937008
```

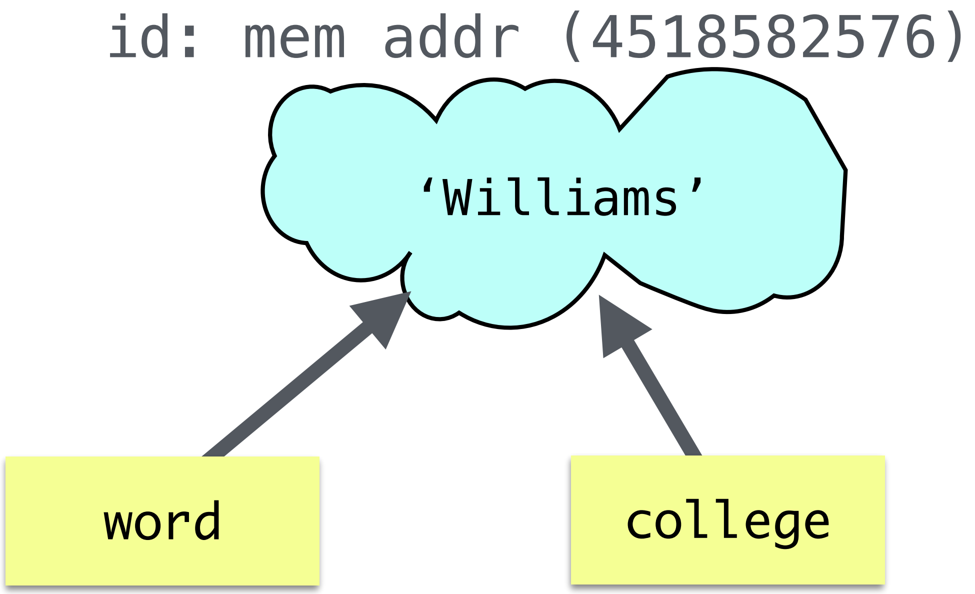


Variable names like `num` point to memory addresses of stored value

Strings are Immutable



```
In [1]: word = "Williams"

In [2]: college = word

In [3]: word == college
Out[3]: True

In [4]: print(id(word), id(college))
        4518582576 4518582576

In [5]: word is college
Out[5]: True
```

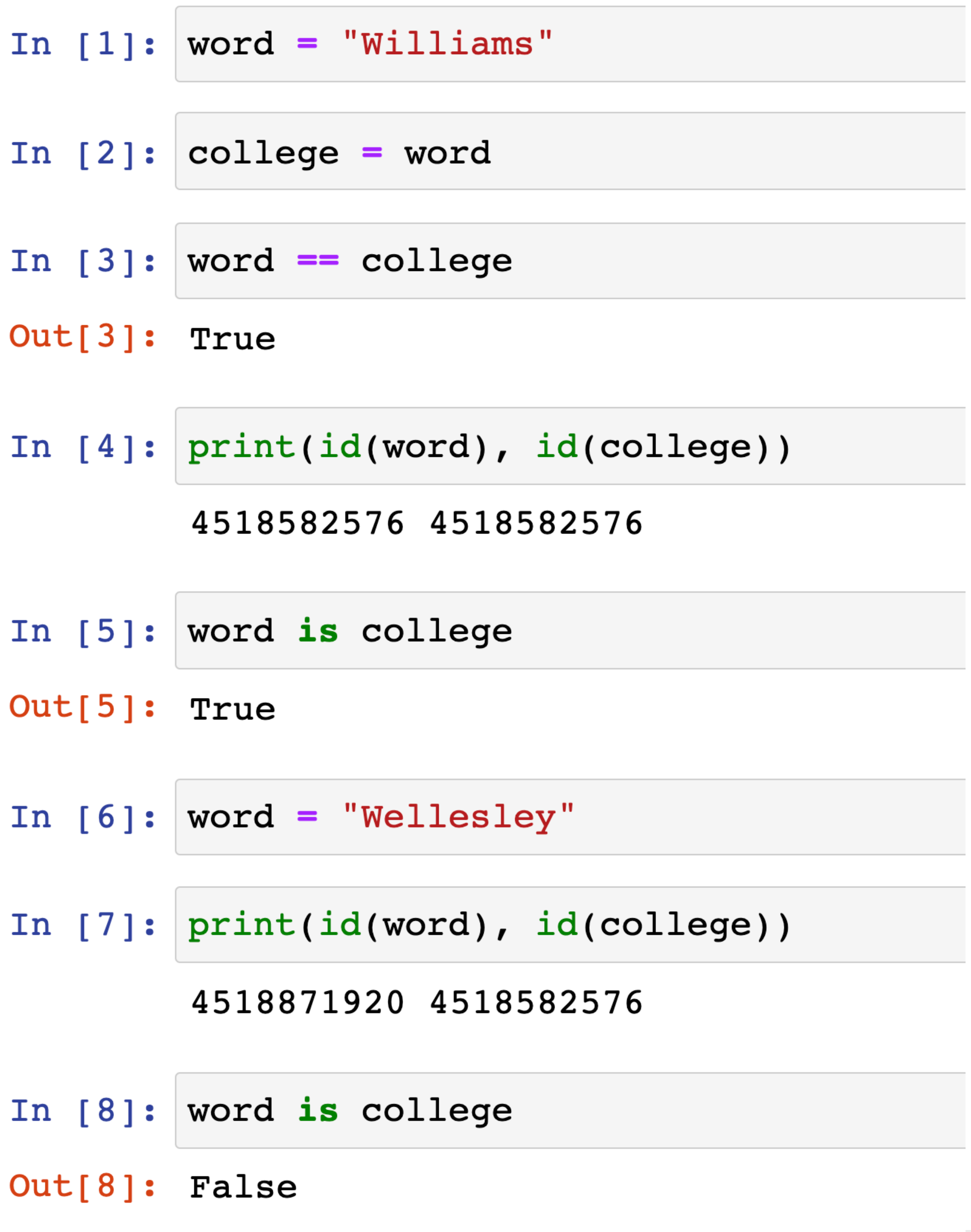


Variable names point to memory addresses of stored value

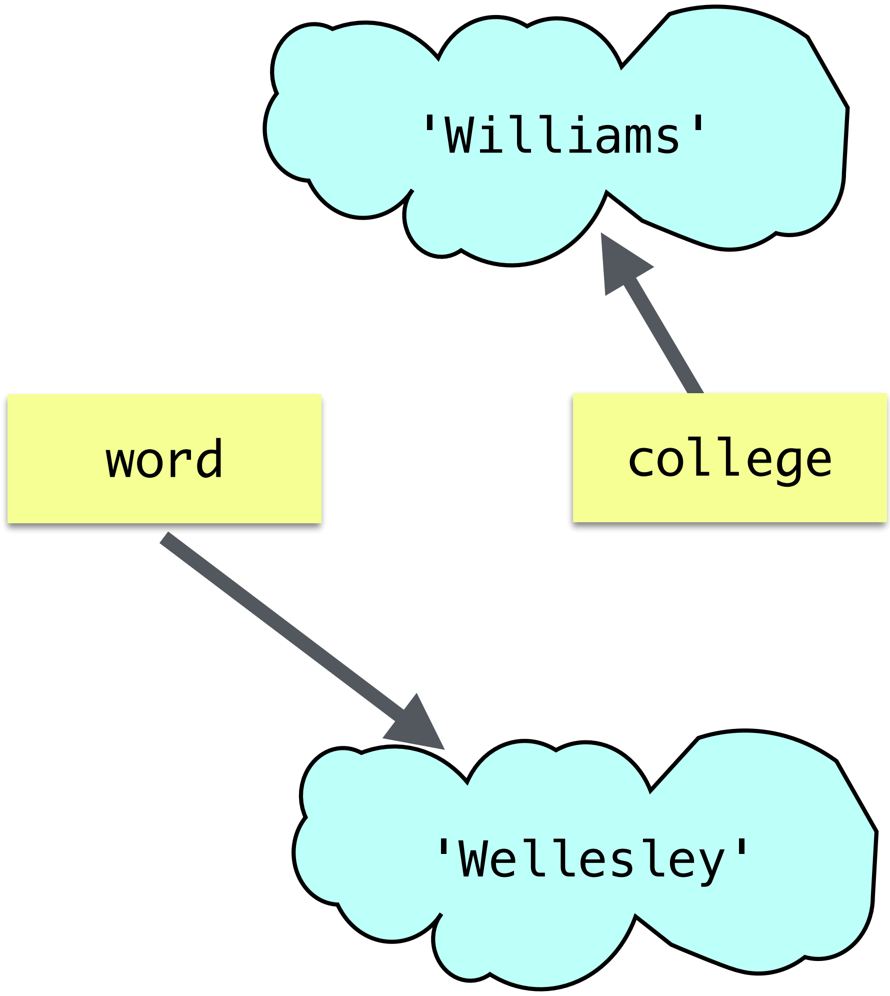
Even though word and college have the same identity and value, if we update one of them, it just assumes a new identity!

Attempts to change an immutable object creates a new object

Strings are Immutable



```
In [1]: word = "Williams"

In [2]: college = word

In [3]: word == college
Out[3]: True

In [4]: print(id(word), id(college))

        4518582576 4518582576

In [5]: word is college
Out[5]: True

In [6]: word = "Wellesley"

In [7]: print(id(word), id(college))

        4518871920 4518582576

In [8]: word is college
Out[8]: False
```



Attempts to change an immutable object creates a new object

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

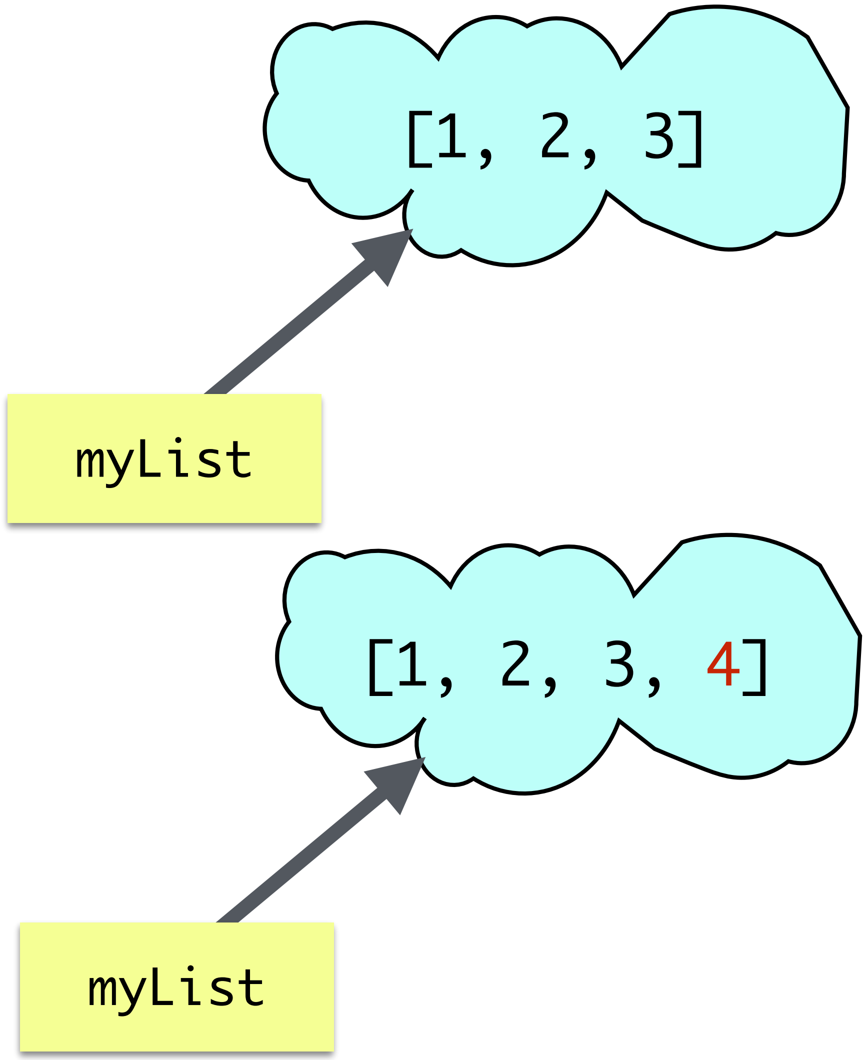
Lists are Mutable

- List values **can** be changed
- We reviewed how we can mutate/change what's in a list using methods
- **Aliasing** happens when the value of one variable is assigned to another variable
  - Can have multiple names for the same object
- The mutability of lists has many implications with respect to ***aliasing***

Lists are Mutable

```
In [1]: myList = [1, 2, 3]

In [2]: id(myList)
Out[2]: 4418551104

In [3]: myList.append(4)

In [4]: id(myList)
Out[4]: 4418551104
```

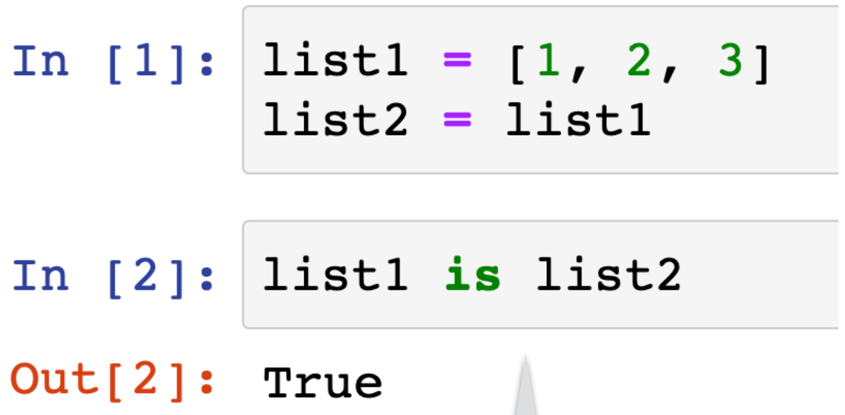
Note: Value changes, identity stays the same



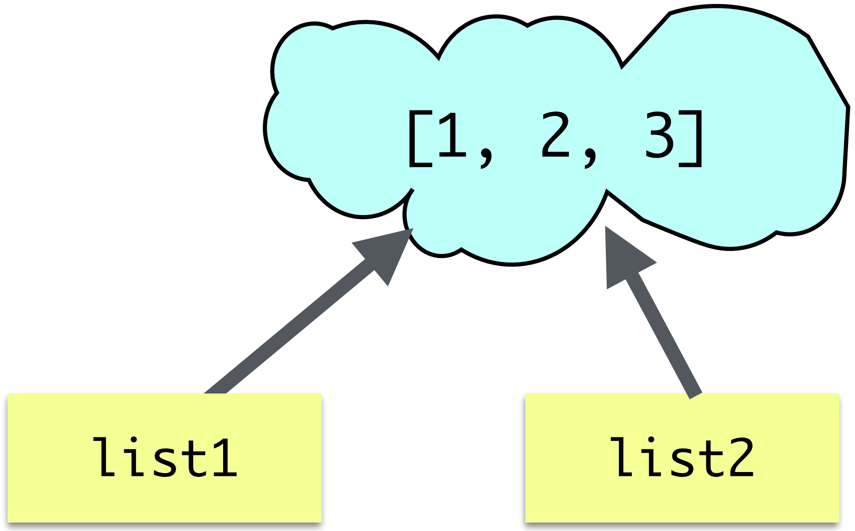
Value of list objects can change, keeping identity the same

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



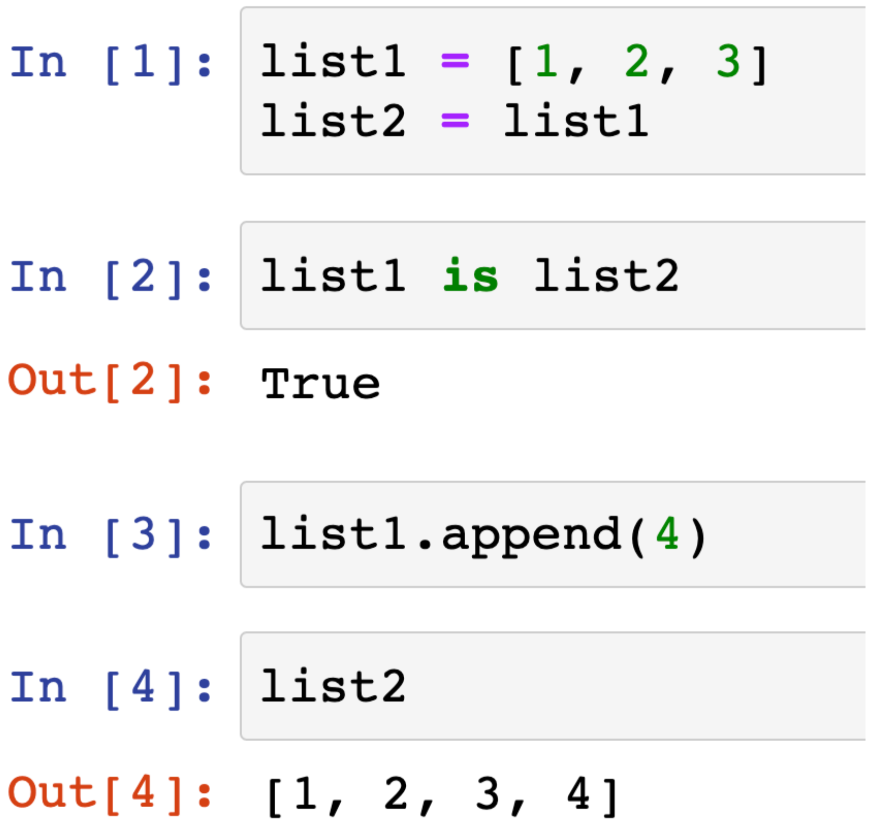
```
In [1]: list1 = [1, 2, 3]
        list2 = list1

In [2]: list1 is list2
Out[2]: True
```



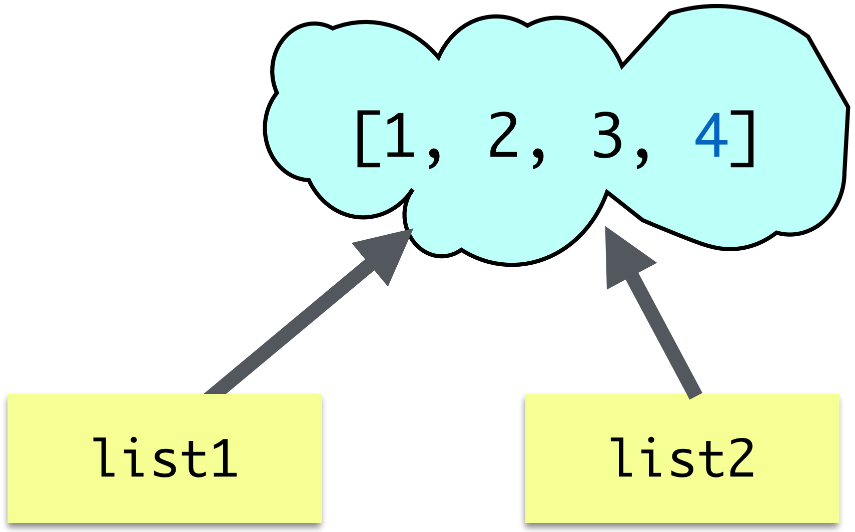
We are not creating a separate copy, but rather creating a **second name** for the original list; **`list2` is an alias of `list1`**

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!

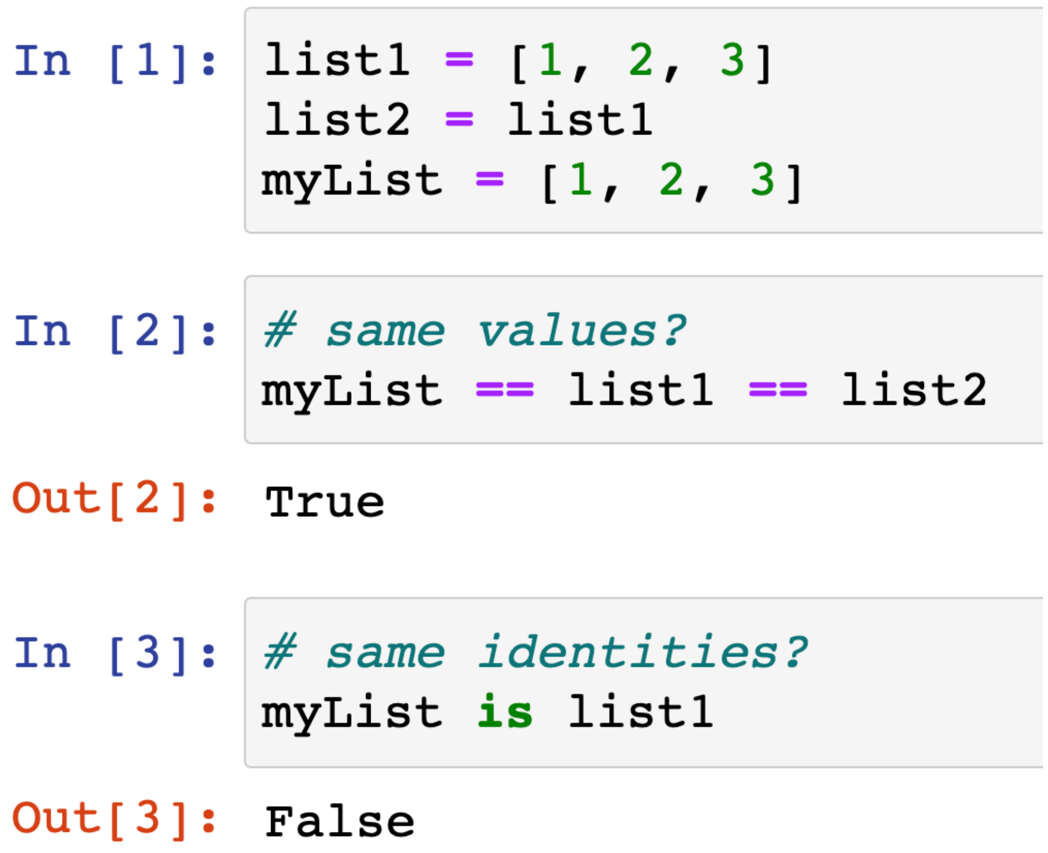


```
In [1]: list1 = [1, 2, 3]
        list2 = list1

In [2]: list1 is list2

Out[2]: True

In [3]: list1.append(4)

In [4]: list2

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

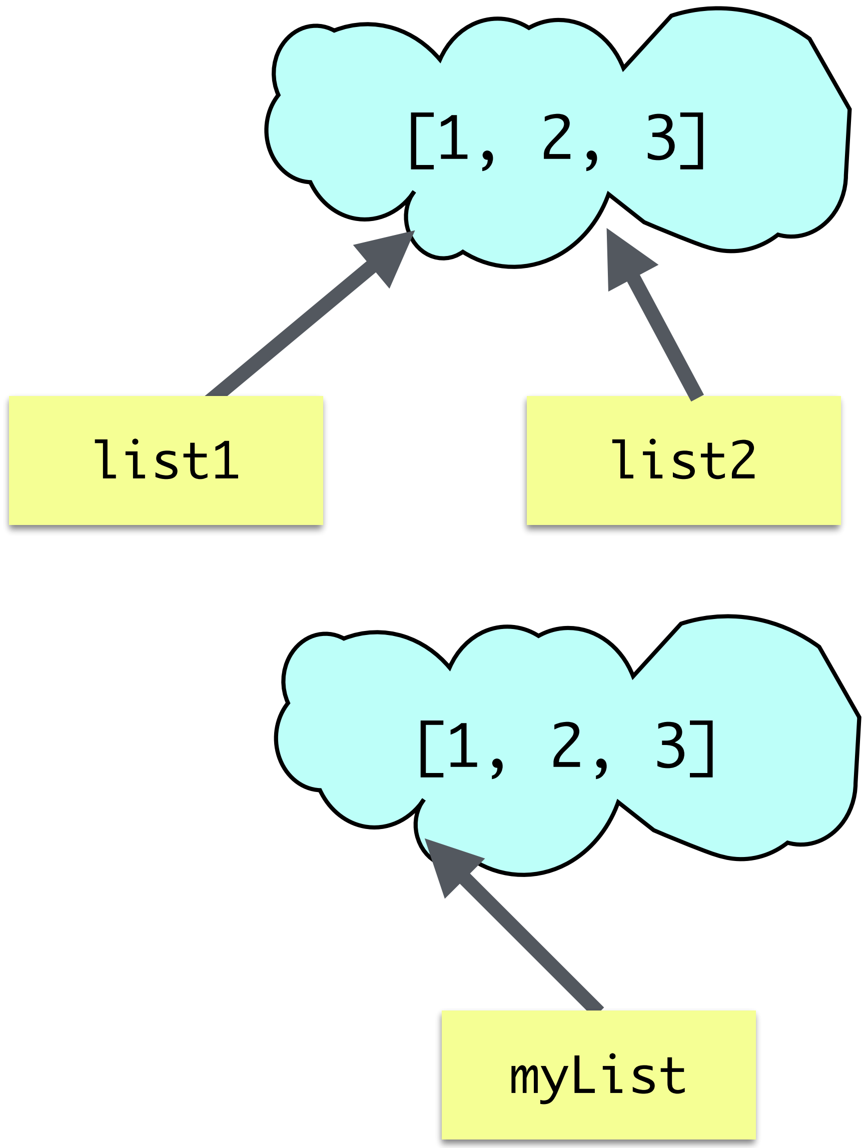


List Aliasing

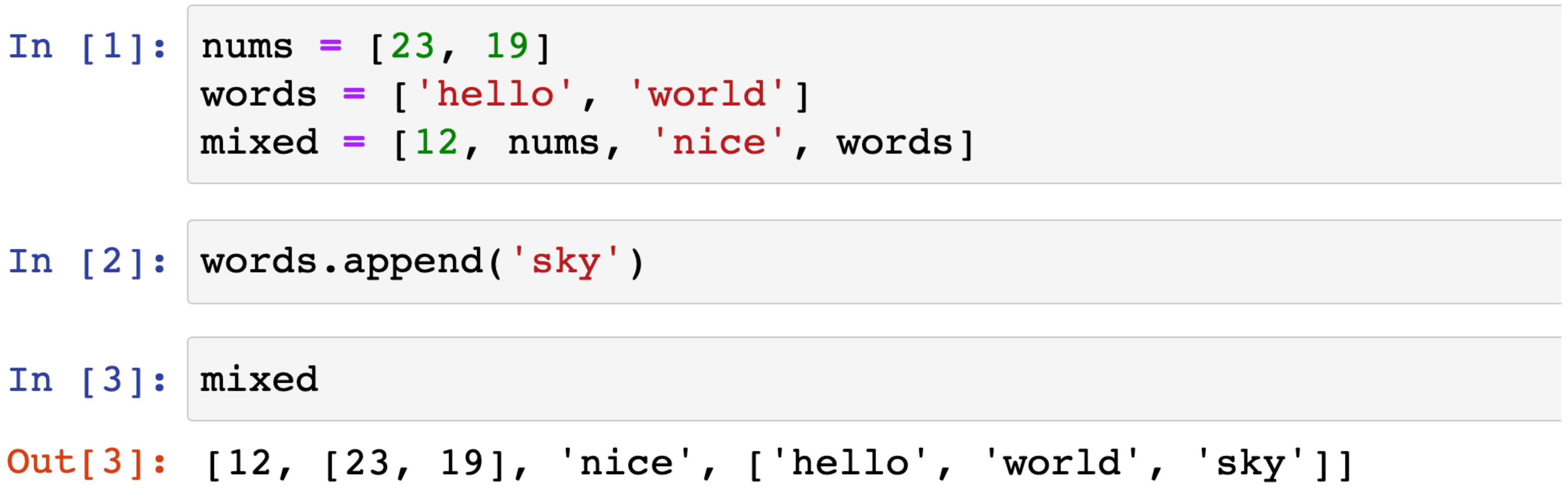
- An assignment to a new variable **creates a new list**



```
In [1]: list1 = [1, 2, 3]
        list2 = list1
        myList = [1, 2, 3]

In [2]: # same values?
        myList == list1 == list2

Out[2]: True

In [3]: # same identities?
        myList is list1

Out[3]: False
```



(Crazy) Aliasing Examples



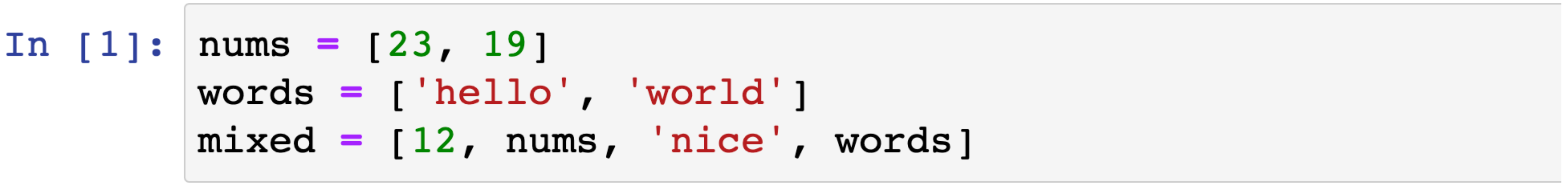
```
In [1]: nums = [23, 19]
        words = ['hello', 'world']
        mixed = [12, nums, 'nice', words]
```

```
In [2]: words.append('sky')
```

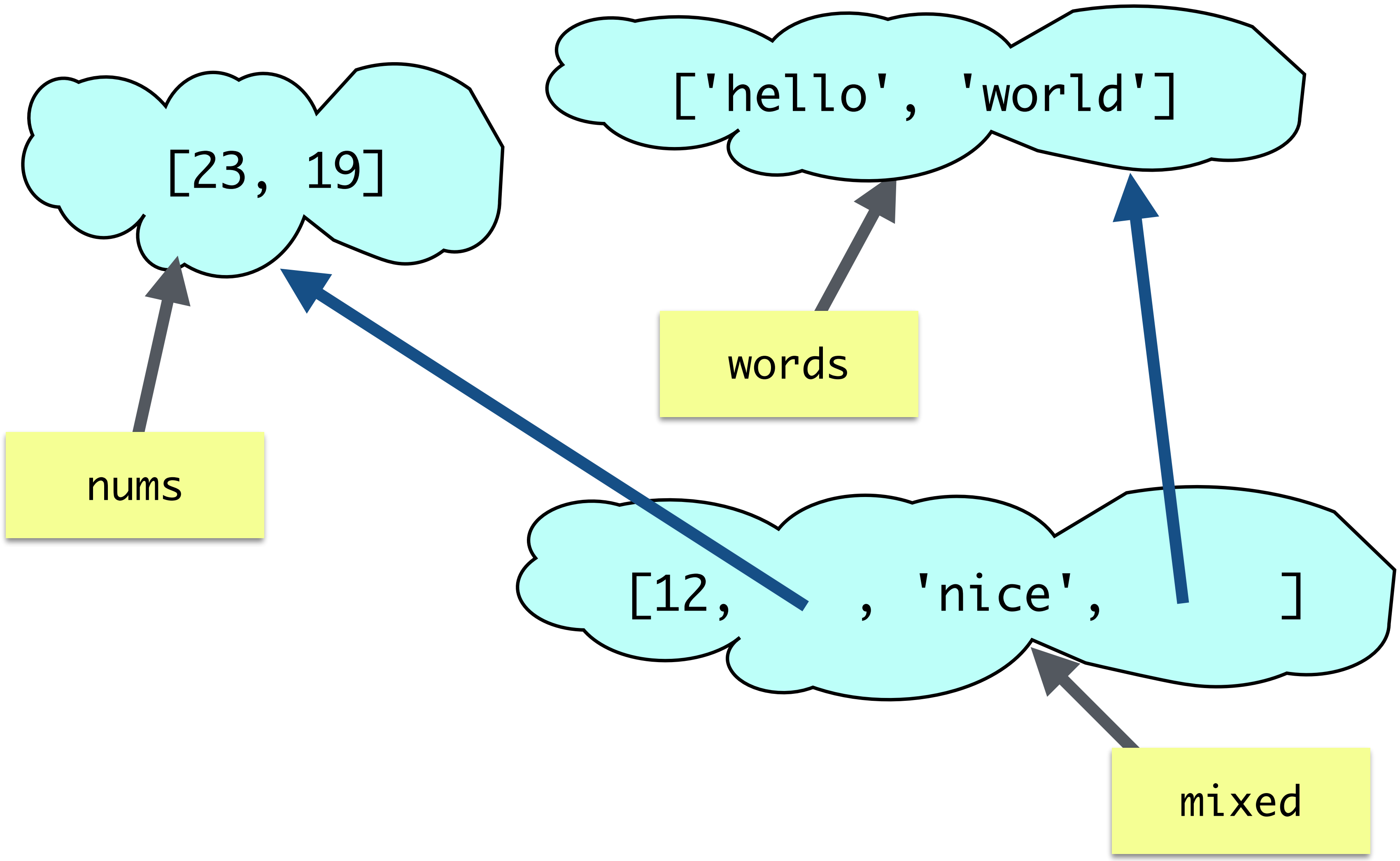
```
In [3]: mixed
```

```
Out[3]: [12, [23, 19], 'nice', ['hello', 'world', 'sky']]
```

(Crazy) Aliasing Examples

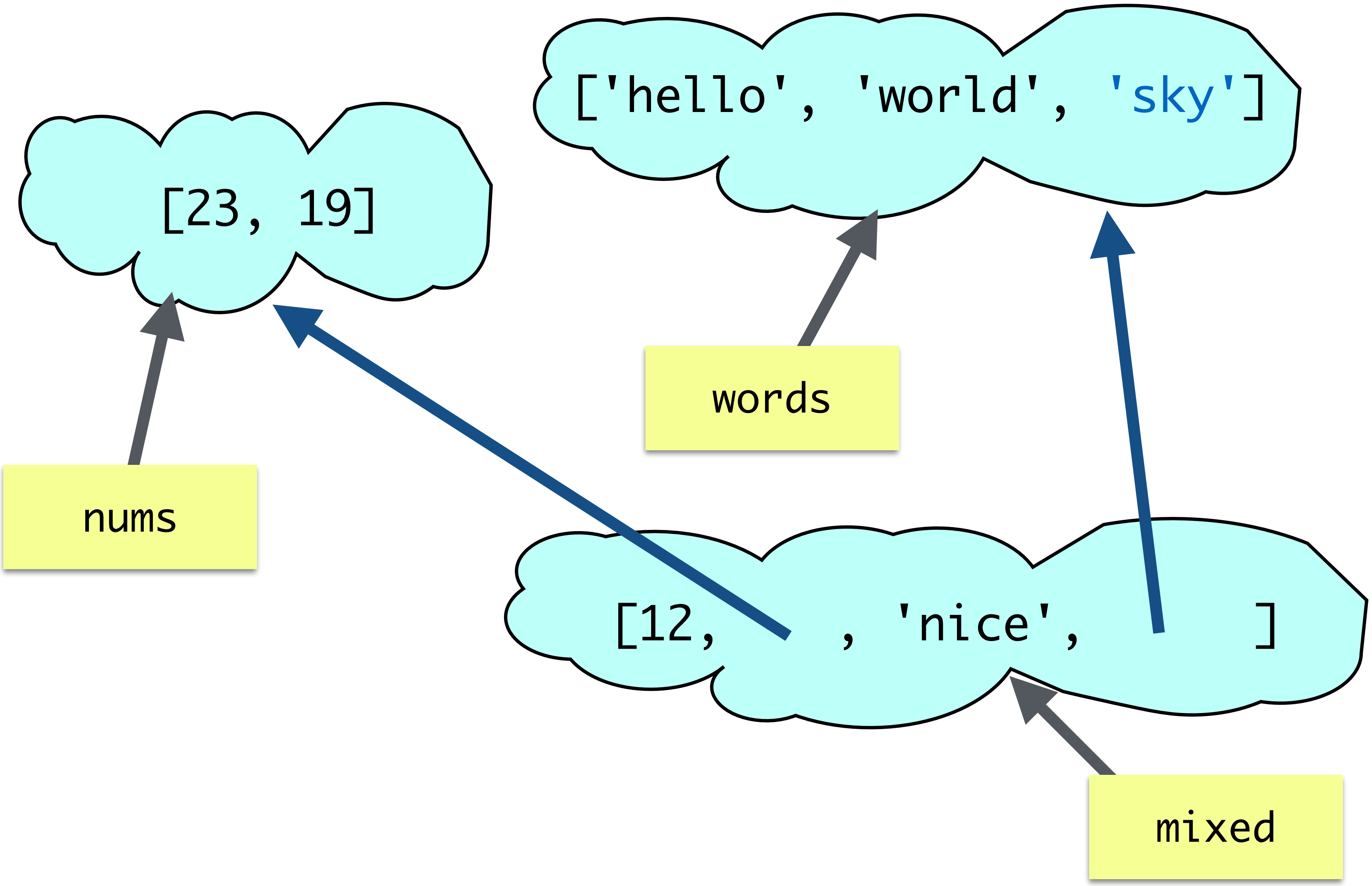


```
In [1]: nums = [23, 19]
        words = ['hello', 'world']
        mixed = [12, nums, 'nice', words]
```

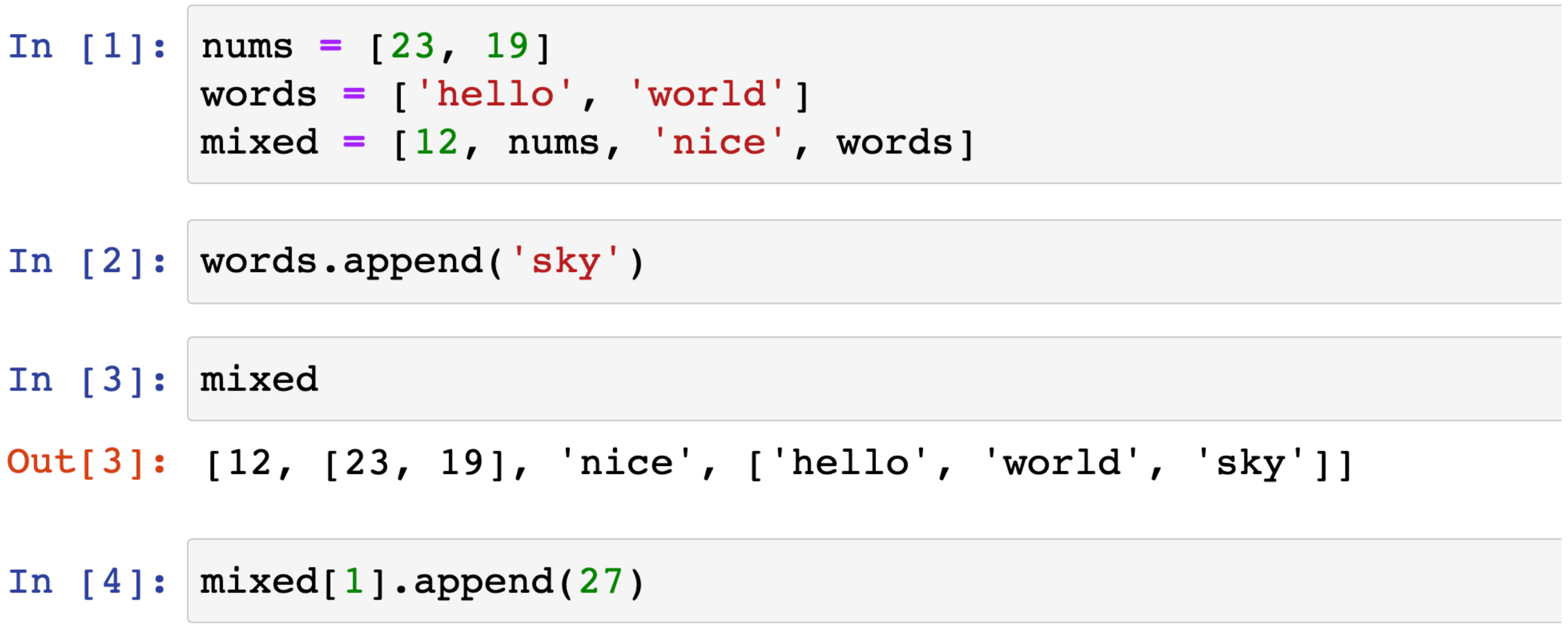


(Crazy) Aliasing Examples

```
In [2]: words.append('sky')
```



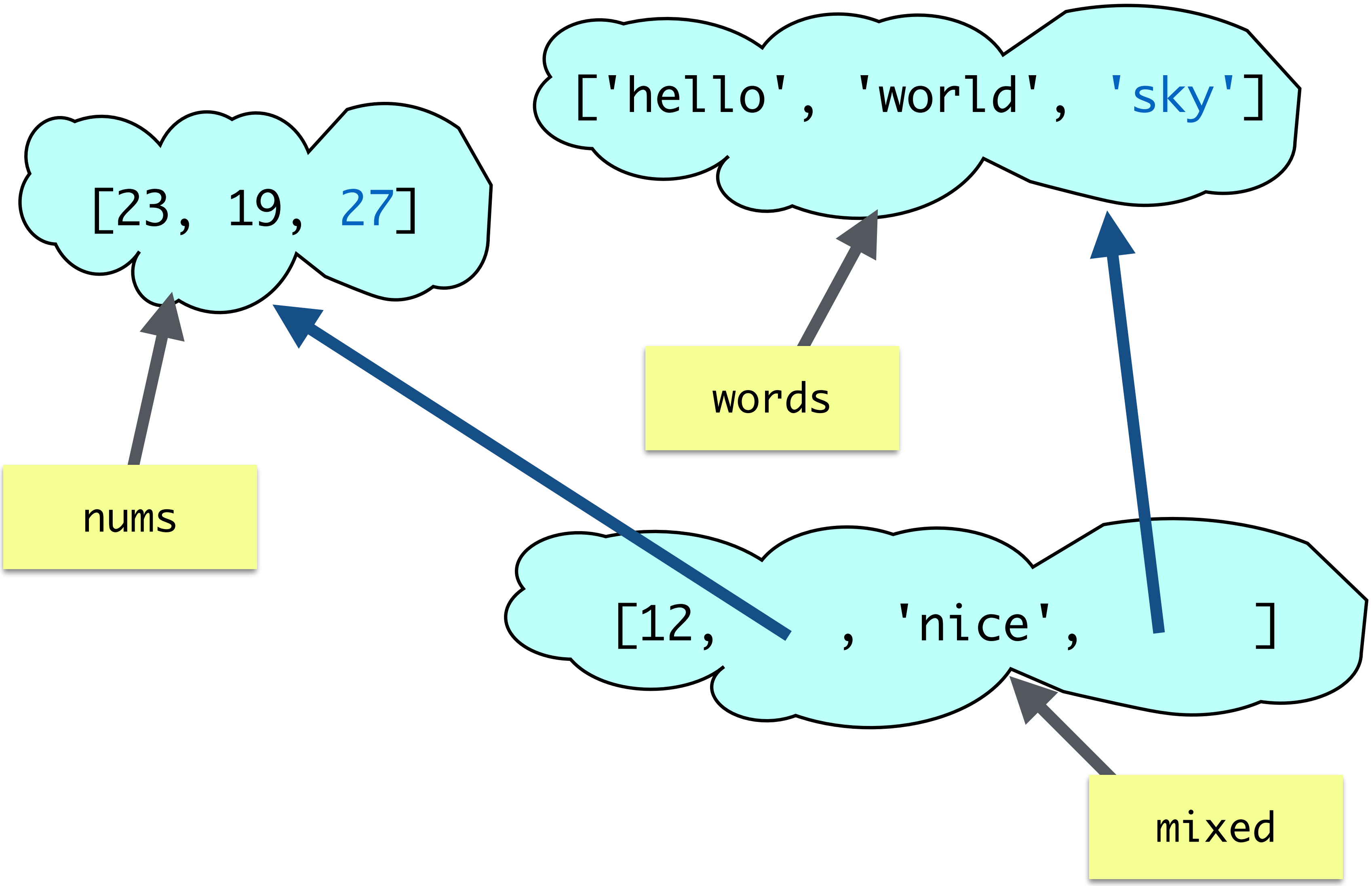
(Crazy) Aliasing Examples



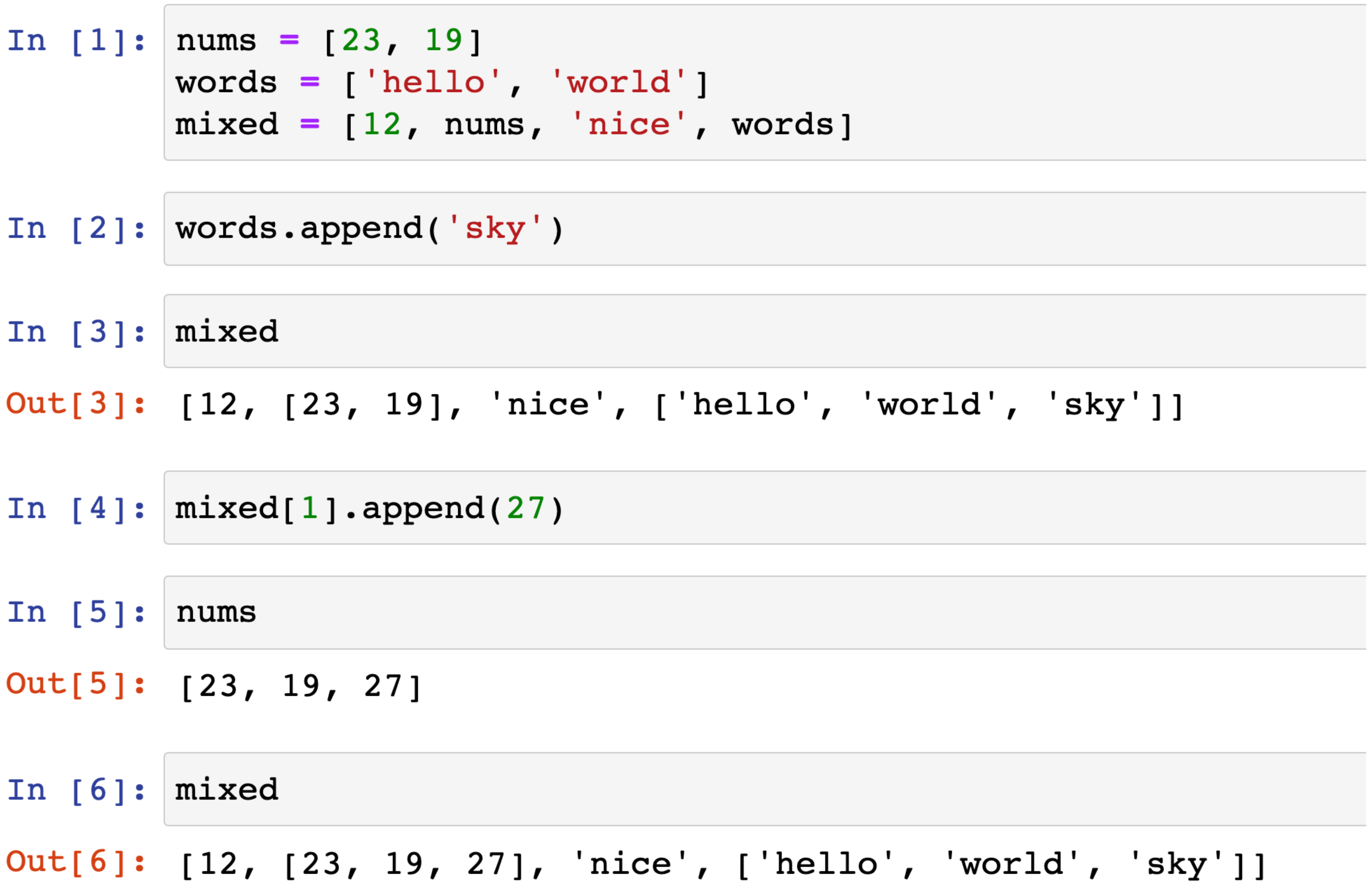
```
In [1]: nums = [23, 19]
        words = ['hello', 'world']
        mixed = [12, nums, 'nice', words]

In [2]: words.append('sky')

In [3]: mixed

Out[3]: [12, [23, 19], 'nice', ['hello', 'world', 'sky']]

In [4]: mixed[1].append(27)
```

(Crazy) Aliasing Examples

```
In [4]: mixed[1].append(27)
```



(Crazy) Aliasing Examples



```
In [1]: nums = [23, 19]
        words = ['hello', 'world']
        mixed = [12, nums, 'nice', words]

In [2]: words.append('sky')

In [3]: mixed

Out[3]: [12, [23, 19], 'nice', ['hello', 'world', 'sky']]

In [4]: mixed[1].append(27)

In [5]: nums

Out[5]: [23, 19, 27]

In [6]: mixed

Out[6]: [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 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 comprehension

```
newList = myList[:]
newList = [ele for ele in myList]
```

  - A (confusing) aside: When using the `+=` operator with lists, it actually calls `append()`! (Always use `myList = myList + [element]` if you want to avoid mutation.)

Moving on…
While Loops

For loops in Python

- **For loops** in Python are meant to iterate directly over a **fixed sequence** of items
    - No need to know the sequence's length ahead of time
- Interpretation of for loops in Python:

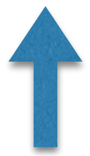
```
for each item in given sequence:
    (do something with item)
```

- Other programming languages (like Java) have for loops that require you to explicitly specify the length of the sequence or a stopping condition
- Thus Python for loops are sometimes called "**for each**" loops
- **Takeaway**: For loops in Python are meant to iterate directly over each item of a given ***iterable*** object (such as a sequence)

What If We Don’t Know When to Stop?

- Stopping condition of for loop: **no more elements in sequence**

```
["A", "cold", "winter", "day"]
```



- What if we don’t know when to stop?
    - Suppose you had to write a program to ask a user to enter a name, repeatedly, until the user enters “quit”, in which case you stop asking for input and print “Goodbye"

While Loops

- For loops iterate over a pre-determined sequence and stop at the end of the sequence
- On the other hand, `while` loops are useful when **we don't know in advance when to stop**
- **while loop syntax:**

```
while (boolean expression evaluates to true):
    # keep repeating the following
    # statements in loop body
    # as long as the loop condition is true
```

- A while loop will keep iterating as long as **the condition in the parentheses is satisfied** (is true) and will halt when the **condition fails to hold** (becomes false)

While Loop Example

- Example of a while loop that depends on user input

```
prompt = 'Please enter a name (type quit to exit): '
name = input(prompt)

while (name.lower() != 'quit'):
    print('Hi,', name)
    name = input(prompt)
print('Goodbye')
```

- See notebook for example tests of this piece of code

While Loop to Print Halves

- Given a number, keep dividing it until it becomes smaller than 0 and print all the "halves"

```
def printHalves(n):
    while n > 0:
        print(n)
        n = n//2

printHalves(100)
```

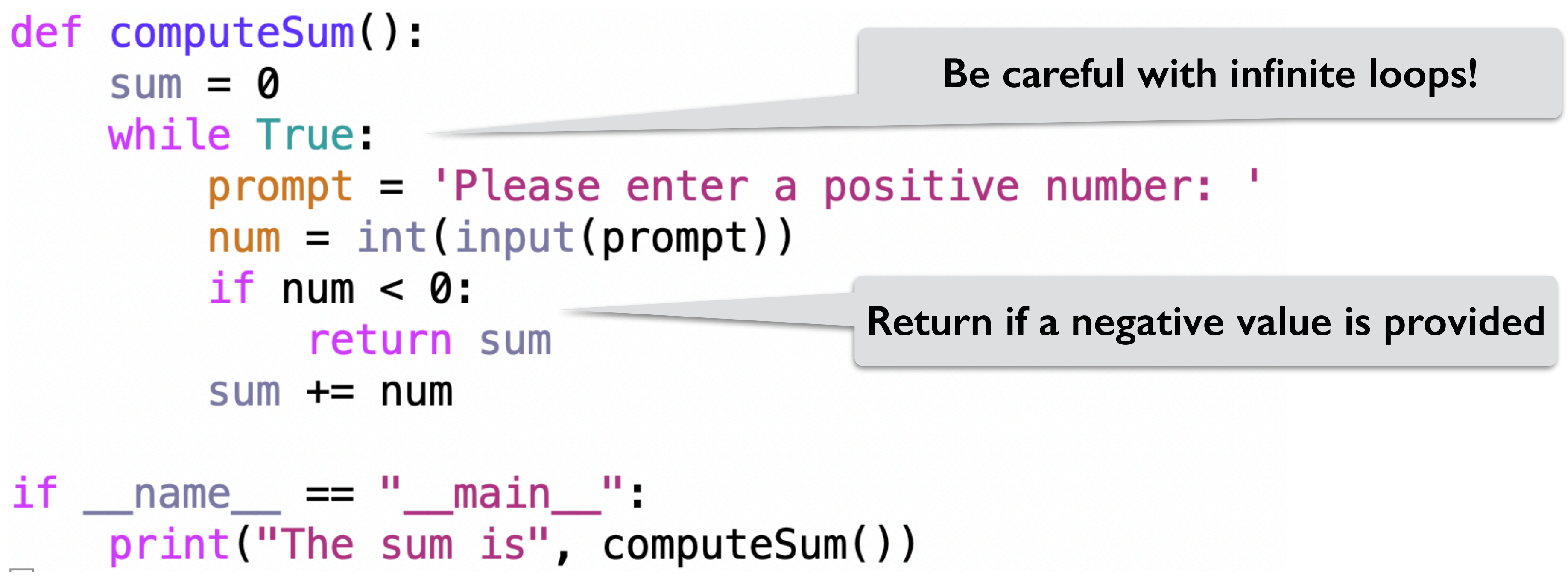
```
100
50
25
12
6
3
1
```

```
def printHalves(n):
    while n > 0:
        print(n)
    n = n//2

printHalves(100)
```

Infinite loop! Indentation matters!

Infinite Loops

- Most of the time, you want to avoid an unintentional **infinite loop**
  - Infinite loops occur when the loop condition **never turns false**
- Occasionally, as in Lab 4, you create an intentional infinite loop
  - This is ok (and sometimes desirable!) as long as **there is a way to exit the loop**
  - A **return** statement will force the loop to exit



```
def computeSum():
    sum = 0
    while True:
        prompt = 'Please enter a positive number: '
        num = int(input(prompt))
        if num < 0:
            return sum
        sum += num

if __name__ == "__main__":
    print("The sum is", computeSum())
```

Be careful with infinite loops!

Return if a negative value is provided