# CS134:
# Aliasing & While Loops

# Announcements & Logistics

- **HW 5** due Mon at 10 pm
  - Last one before midterm! This one is a little tricky!
- **Lab 4 Part 1** due today/tomorrow at 10pm
  - We'll send automated feedback about Part 1 on Friday
  - **Part 2** due next Wed/Thur at 10 pm
  - Lab "style suggestions" posted (see Friday on calendar)
- Student help hours during Reading Days:
  - XXX
- **Midterm reminder**: Thur Oct 20: 6 - 7:30 pm or 8 - 9:30 pm
- **Midterm review**: Tue Oct 18: 8 - 9:30 pm

**Do You Have Any Questions?**

# Last Time

- 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

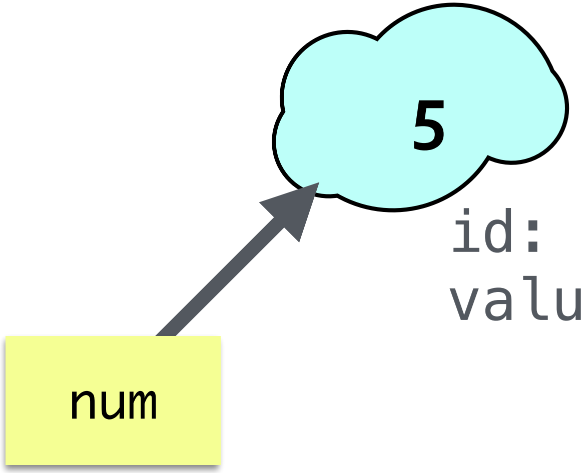
- Continue discussing **aliasing** and **mutability** in Python
- Discuss while loops
  - Needed for ranked-choice voting on Lab 4 Part 2

# Mutability & Aliasing

# Recap: Value vs Identity

- An **object's identity** never changes once it has been created
  - The `id()` function returns an object's identity (or address)
  - Compare with `is` operator
- An **object's value** is the value assigned to the object when it is created
  - Objects whose values can change are **mutable**; objects whose values cannot change are called **immutable**
  - Compare with `==` operator

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



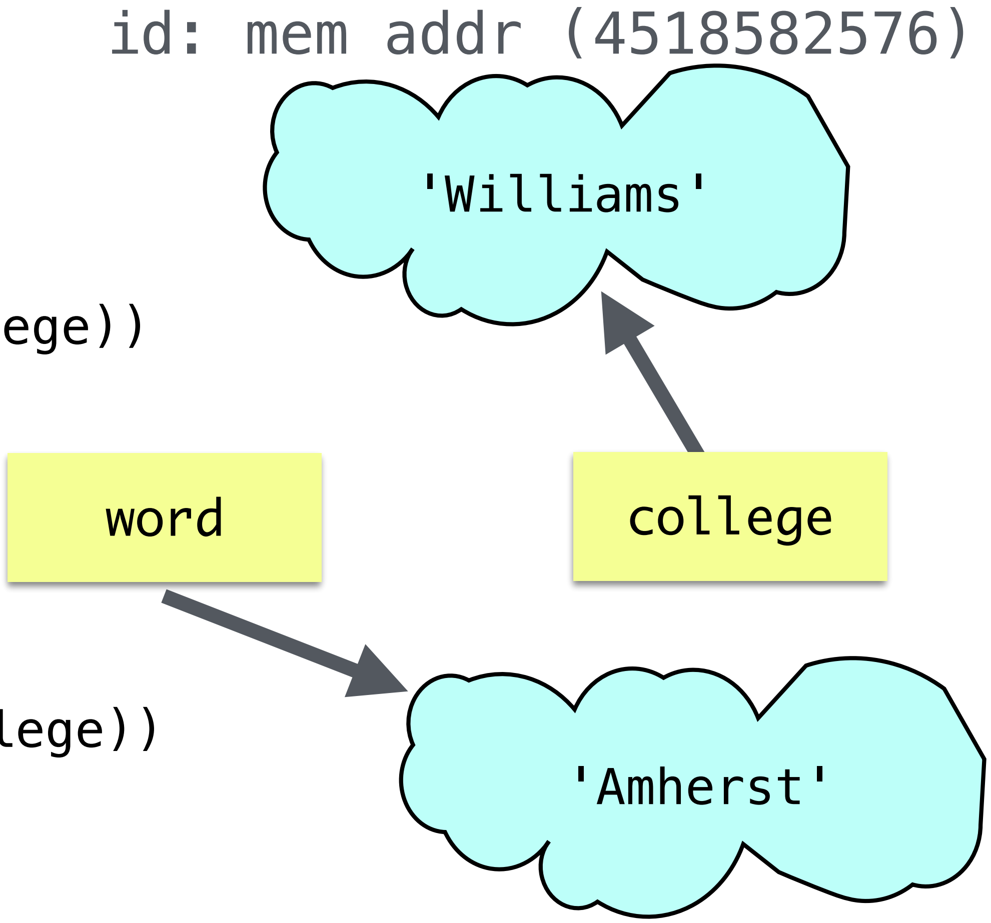
Memory address

```
id: 4486937008
value: 5
```

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

# 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
```



id: mem addr (4518582576)

'Williams'

word

college

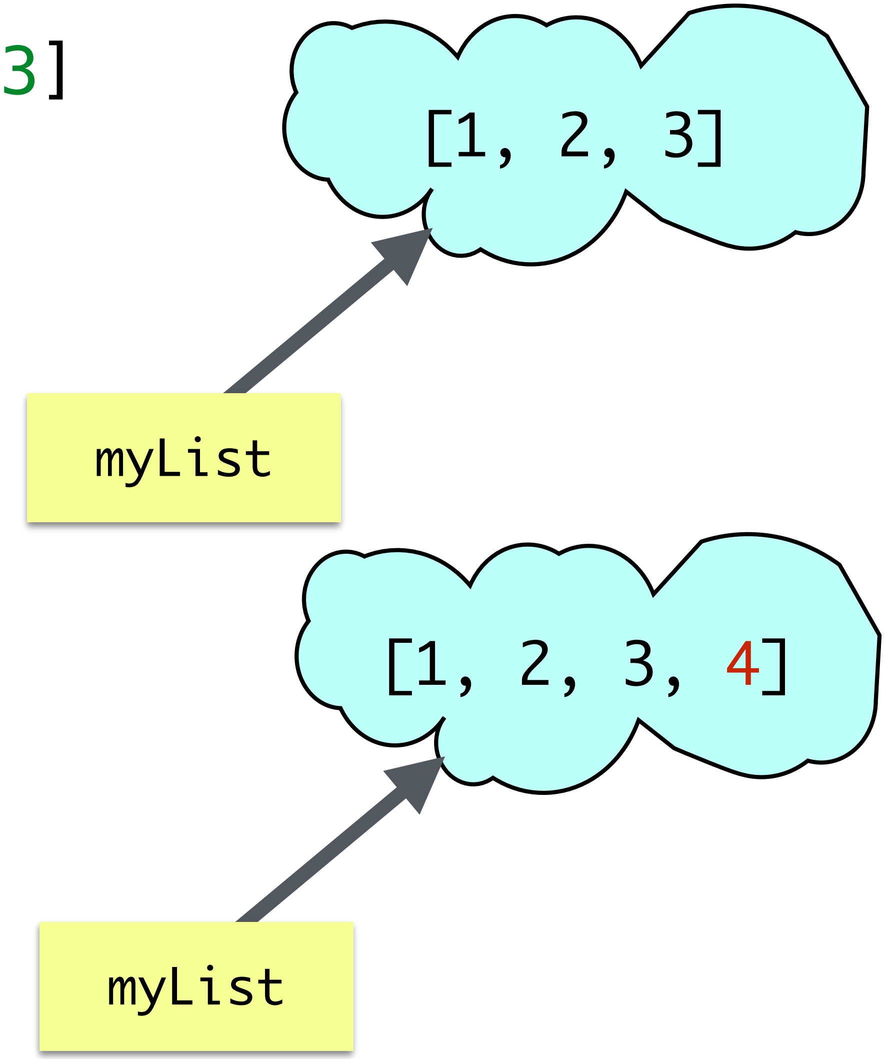
'Amherst'

Even though word and college initially 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**

# Lists are Mutable

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

>>> myList.append(4)
>>> id(myList)
4418551104
```



**Note**: Value changes, identity stays the same

**Value of list objects can change, keeping identity the same**

# 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!

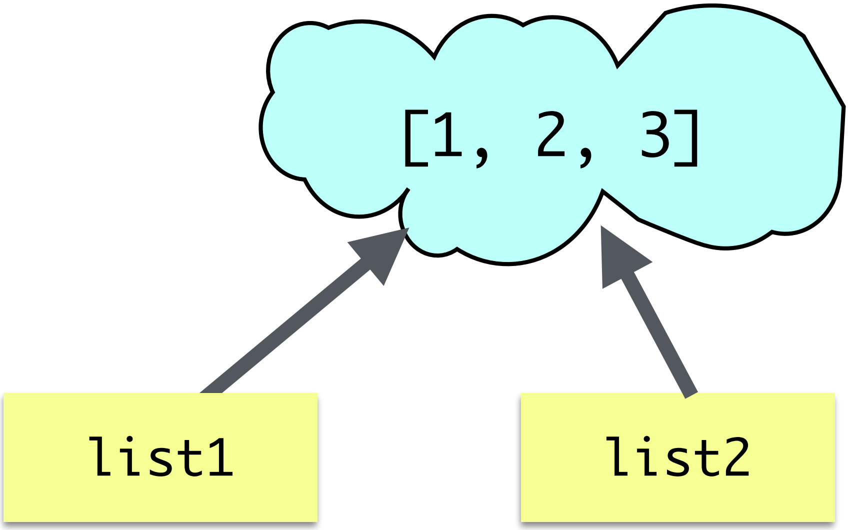
# List Aliasing

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
```

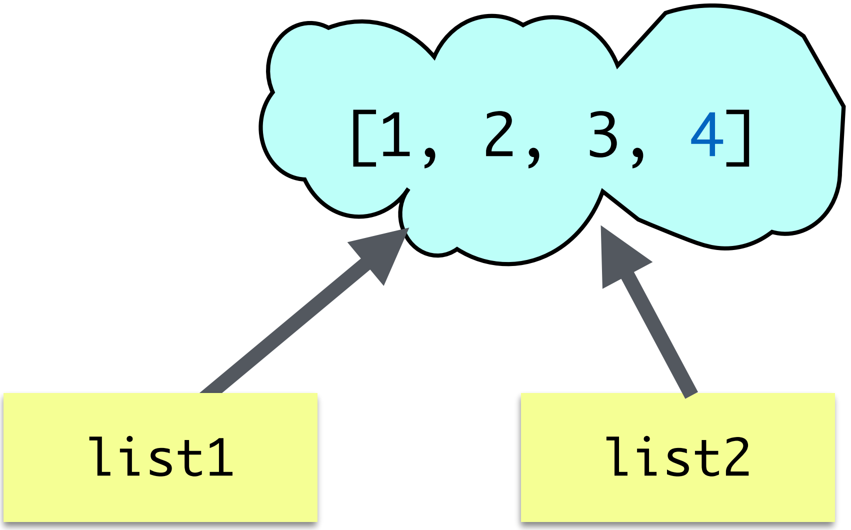


We are not creating a separate copy, but rather creating a **second name** for the original list; **`list2` is an alias of `list1`**

# 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!

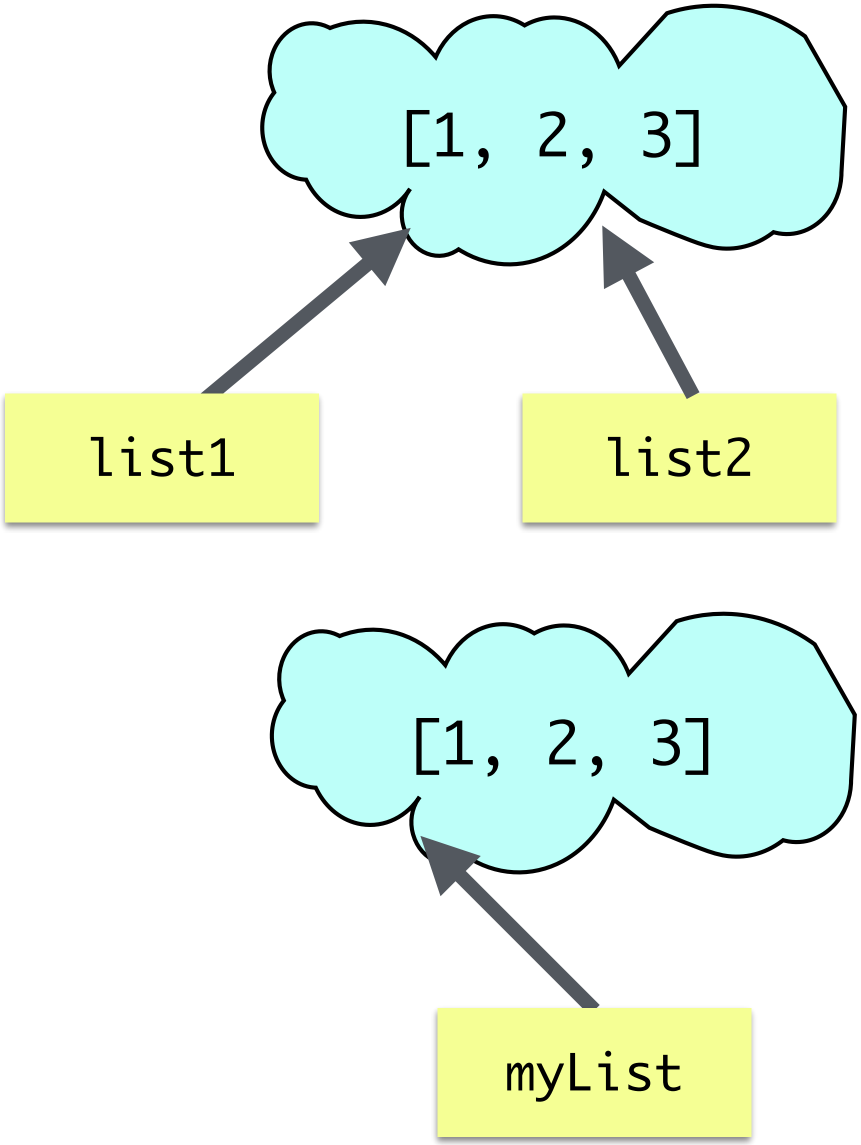
```
>>> list1 = [1, 2, 3]
>>> list2 = list1

>>> list1 is list2
True

>>> list1.append(4)
>>> list2

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



# List Aliasing

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

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

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

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



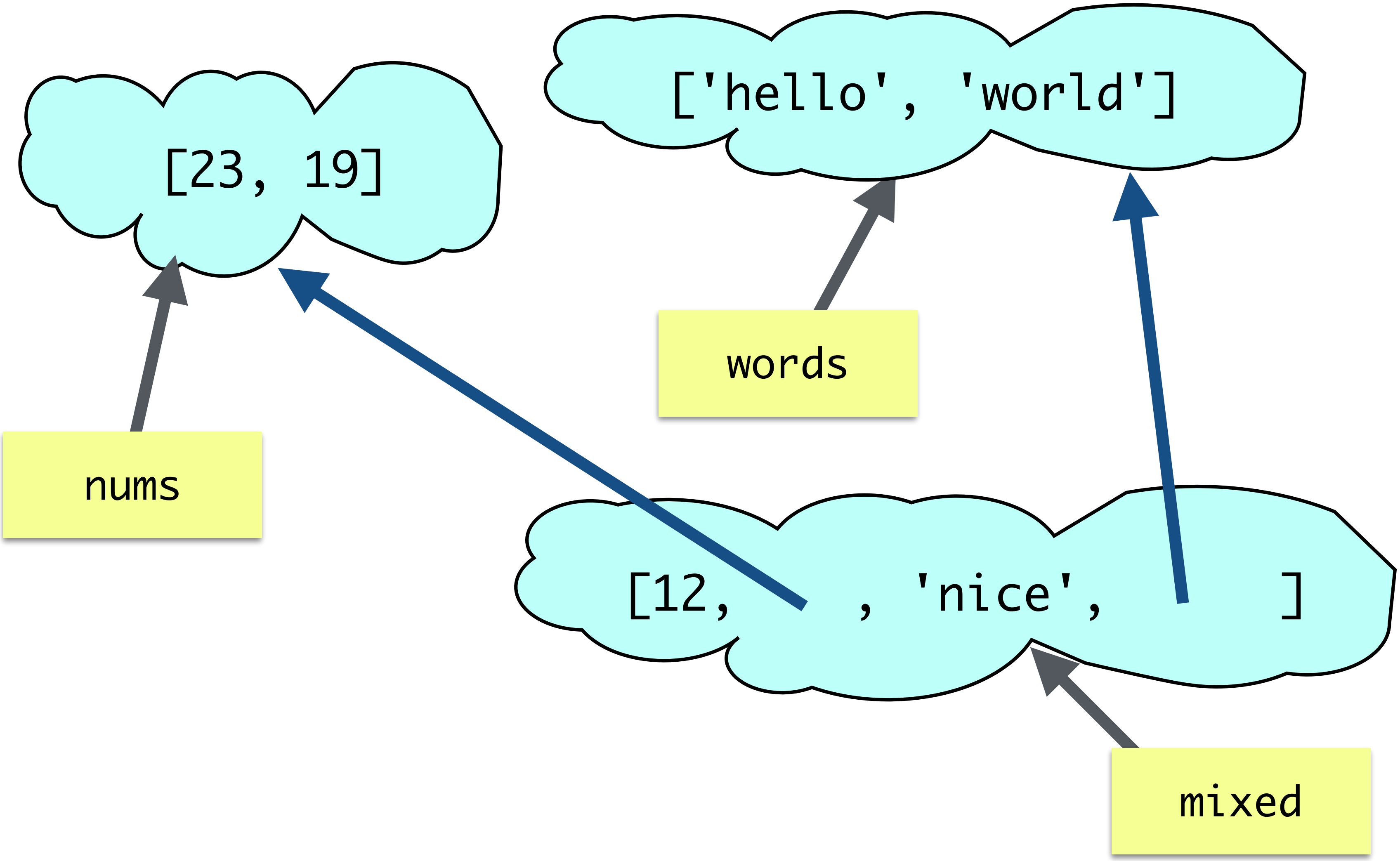
# (Crazy) Aliasing Examples

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

>>> words.append("sky")
>>> mixed
```

???

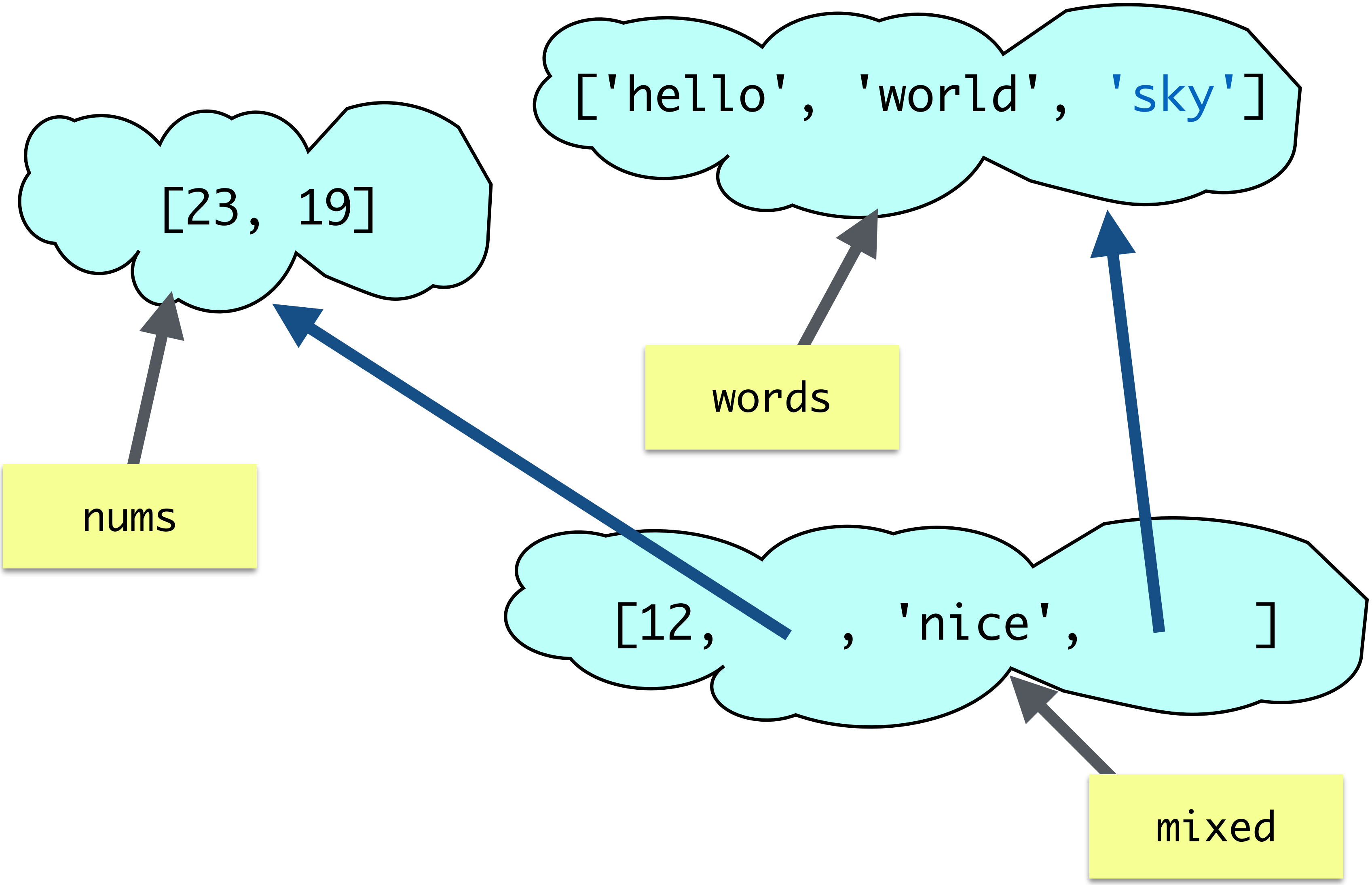
# (Crazy) Aliasing Examples

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



# (Crazy) Aliasing Examples

```
>>> words.append("sky")
```



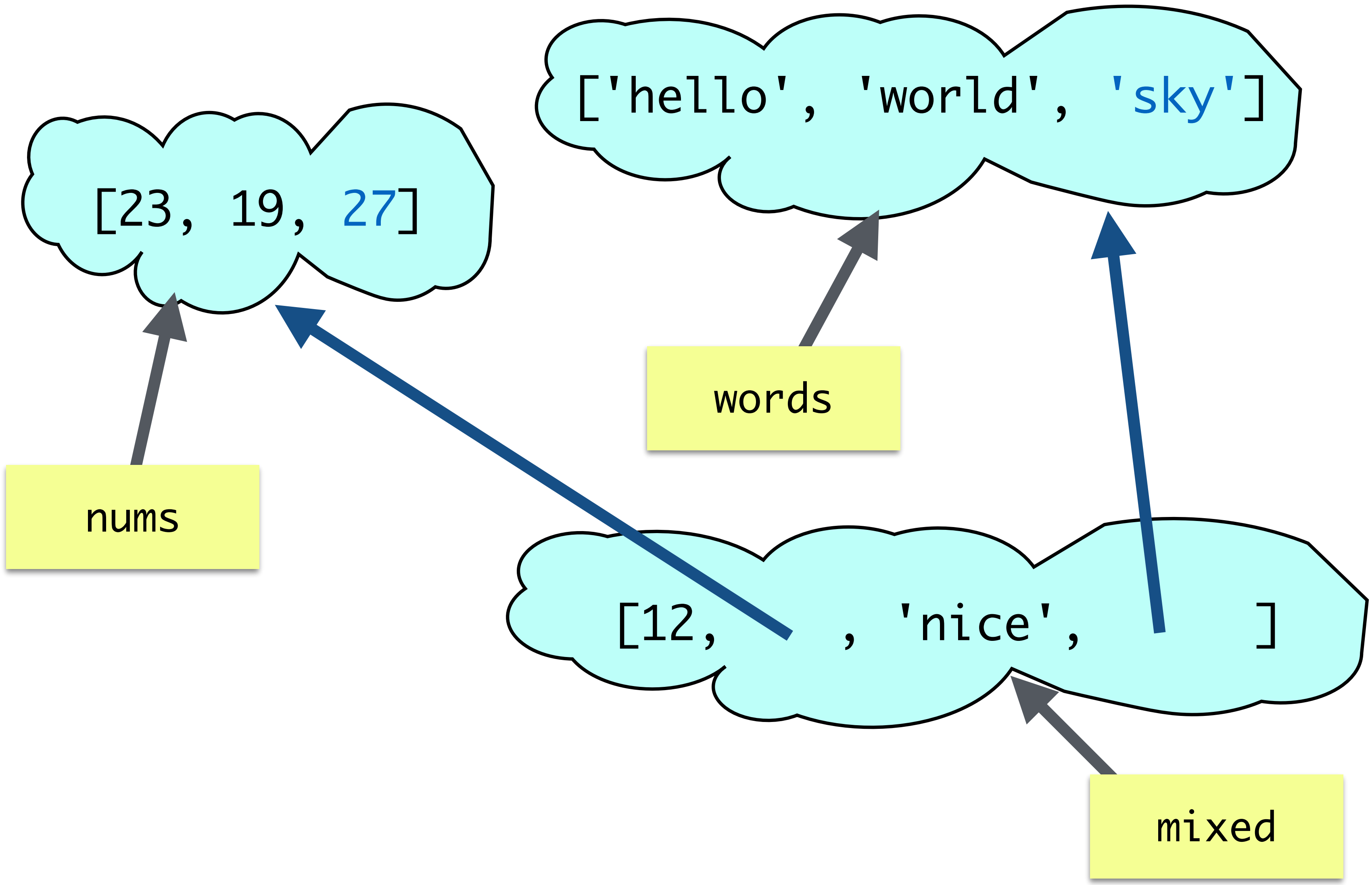
# (Crazy) Aliasing Examples

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

>>> words.append("sky")
>>> mixed
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]

>>> mixed[1].append(27)
```

???

# (Crazy) Aliasing Examples

```
>>> mixed[1].append(27)
```



# (Crazy) Aliasing Examples

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

>>> words.append("sky")
>>> mixed
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]

>>> mixed[1].append(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 comprehension

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

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

# While Loops

# For loops in Python

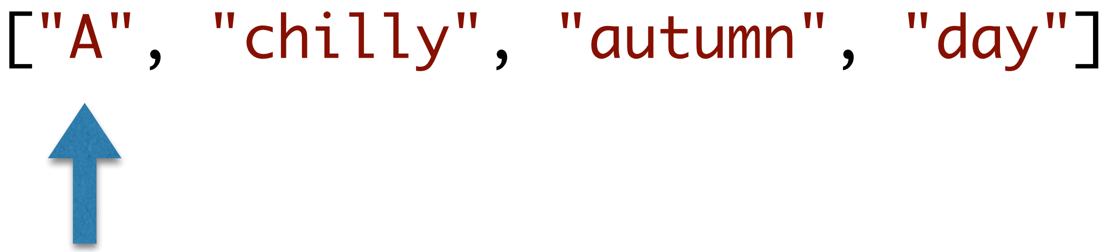
- **For loops** in Python are meant to iterate directly over a **fixed sequence** of items
    - No need to know the sequence's length ahead of time
- Interpretation of for loops in Python:

```
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", "chilly", "autumn", "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**

# The end!