CS 134
Lists & Mutability

Announcements & Logistics

- **Lab 3** graded feedback will be returned soon: make sure to review it
- **Lab 4 Part 1** is due tonight/tomorrow at 11 pm
  - We will run some tests and return automated feedback
  - **Part 2** is due next week
- **Homework 5** will be posted later today, due Mon at 11pm
  - (Probably) Last HW before the midterm
- **Midterm**: Thur Mar 17 in evening
  - **Review session**: Tue Mar 15 in evening (room TBD)

Do You Have Any Questions?

How to Read GradeSheet.txt

```
GRADE SHEET FOR CS134 LAB 3 ("Building a Pyt

Requirements of this lab:
   0. The toolbox (wordTools.py)
      + Module documentation string ("""..."""" at top of module)
      ~ Fixed doctest for canon
      - Some other item
   1. ...

Grade:  B

Comments from Graders:


Specific comments you should pay attention to.
```

+ Good
~ Okay
- Needs work

How to Read Feedback in Code

```
...

#$ Combine these cases with and:
#$ if isIsogram(word) and len(word) == 7:
if isIsogram(word):
    if len(word) == 7:
        count += 1

...
```

How to Read TestResults.txt

```
Summary
=======
test0_wordToolsModified (test1.BasicTests) ........ ok
...
test21_isIsogram (test2.WordToolTests) ............. failed
...


Details
=======

Failed: test21_isIsogram (test2.WordToolTests)
----------------------------------------------------------------------
    55     def test21_isIsogram(self):
    56         val = isIsogram('Aaron')
    57         self.assertEqual(val, False)
----------------------------------------------------------------------
Line 57 resulted in:
  AssertionError: True != False
```

Last Time

- Reviewed **CSV file reading** and accessing **lists of lists**
- Used our knowledge about lists and loops to analyze "interesting" properties of our data
    - Focused on maintaining the state of variables when looping, and how to update state based on conditionals
    - Example functions: `mostVowels, leastVowels`

Today's Plan

- Learn about writing and appending to files
- Review useful list methods that modify the list:
  - `.append()`, `.extend()`, `.insert()`, `.remove()`, `.pop()`, `.sort()`
- Discuss implications of **mutability** in Python

An Aside: Writing to Files

- We know how to ***read from*** files
- We can also ***write to*** files
- We can write all the results that we are computing into a file. To open a ***new*** file for writing, we use `open` with the mode '`w`'.
- Use `.write()` file method to add a string to a file

```
In [65]: fYears = len(yearList(allStudents, 25))
         sophYears = len(yearList(allStudents, 24))
         jYears = len(yearList(allStudents, 23))
         sYears = len(yearList(allStudents, 22))
         mostVowelNames = ', '.join(mostVowels(firstNames))
         leastVowelNames = ', '.join(leastVowels(firstNames))

         with open('studentFacts.txt', 'w') as sFile:
             sFile.write('Fun facts about CS134 students:\n')# need newlines
             sFile.write('Students with most vowels in their name: {}.\n'.format(mostVowelNames))
             sFile.write('Students with least vowels in their name: {}.\n'.format(leastVowelNames))
             sFile.write('No. of first years in CS134: {}.\n'.format(fYears))
             sFile.write('No. of sophmores in CS134: {}.\n'.format(sophYears))
             sFile.write('No. of juniors in CS134: {}\n'.format(jYears))
             sFile.write('No. of seniors in CS134: {}\n'.format(sYears))
```

String method useful in printing

Format Printing for Python Strings

- A convenient way to build strings with particular form is to use the `.format()` string method

  Syntax: `myString.format(*args)`

  `*args` means it takes zero or more arguments

- For every pair of braces (`{}`), format ***consumes*** one argument
- Argument is ***implicitly converted to a string*** and concatenated with the remaining parts of the format string
- Especially useful in printing to files

```
In [8]: "Hello, you {} world{}".format("silly",'!')  # creates a new string
Out[8]: 'Hello, you silly world!'

In [9]: print("Hello, {}.".format("you silly world!"))
        Hello, you silly world!.
```

Appending to Files

- If a file already has something in it, opening it in `w` mode again will erase all of its past contents
- We can also ***append*** something to an ***existing*** file without erasing the contents. To do that we open in append `a` mode.

```
with open('studentFacts.txt', 'a') as sFile:
    sFile.write('Goodbye.\n')
```

```
In [63]: cat studentFacts.txt
```

```
Fun facts about CS134 students:
Students with most vowels in their name: Adelaide, Giulianna.
No. of first years in CS134: 48.
No. of sophmores in CS134: 19.
No. of juniors in CS134: 7
No. of seniors in CS134: 3
Goodbye.
```

List Mutability

A quick review of old and new methods that modify a list:

`.append()`, `.extend()`,
`.pop()`, `.insert()`, `.remove()`, `.sort()`

Direct Modification: Element Assignment

`myList[index] = item` : though not a method, an assignment to a specific index can modify a list directly

Example.

```
myList[1] = 7  # assign 7 to index 1 of myList
```

myList Before	myList After
[1, 2, 3, 4]	[1, 7, 3, 4]

append()

`myList.append(item)` : appends item to end of list

Example.

```
myList.append(5)  # insert 5 at the end of the list
```

myList Before	myList After
`[1, 7, 3, 4]`	`[1, 7, 3, 4, 5]`

extend()

`myList.extend([itemList])`: appends all the items in `itemList` to the end of `myList`.

Example.

```
myList.extend([6, 8]) # insert both 6 and 8 at
the end of the list
```

myList Before	myList After
`[1, 7, 3, 4, 5]`	`[1, 7, 3, 4, 5, 6, 8]`

pop()

`myList.pop(index):` Removes the item at **a given index** (`int`) **and returns it.** If no index is given, by default, `pop()` removes and returns the **last item** from the list.

Example.

```
val = myList.pop(3)
```

returns

```
val = 4
```

`myList Before`

```
[1, 7, 3, 4, 5, 6, 8]
```

`myList After`

```
[1, 7, 3, 5, 6, 8]
```

pop()

`myList.pop(index):` Removes the item at **a given index** (`int`) **and returns it.** If no index is given, by default, `pop()` removes and returns the **last item** from the list.

Example.

No Index

`val = myList.pop()` → **returns** → `val = 8`

myList Before	myList After
`[1, 7, 3, 5, 6, 8]`	`[1, 7, 3, 5, 6]`

insert()

`myList.insert(index, item):` inserts item at index (`int`) in `myList`, all items to the right of index shift over to make room

Example.

```
myList.insert(0,11)  # insert 11 at index 0
```

myList Before	myList After
[1, 7, 3, 5, 6]	[11, 1, 7, 3, 5, 6]

insert()

`myList.insert(index, item):` inserts item at index (`int`) in `myList`, all items to the right of index shift over to make room

inserting at an index out of range

Example.

```
myList.insert(10,12)  # insert 12 at index 10
```

myList Before	myList After
`[11, 1, 7, 3, 5, 6]`	`[11, 1, 7, 3, 5, 6, 12]`

remove()

`myList.remove(item):` removes first occurrence of `item` from `myList`, all items to the right of removed item shift to the left by one

(Unlike pop(), item is not returned!)

Example.

```
myList.remove(12)    # remove 12 from myList
```

myList Before	myList After
[11, 1, 7, 3, 5, 6, 12]	[11, 1, 7, 3, 5, 6]

sort()

`myList.sort(item):` sorts the list in place in ascending order

Example.

```
myList.sort()   # sort by mutating myList
```

`myList` Before

`[11, 1, 7, 3, 5, 6]`

`myList` After

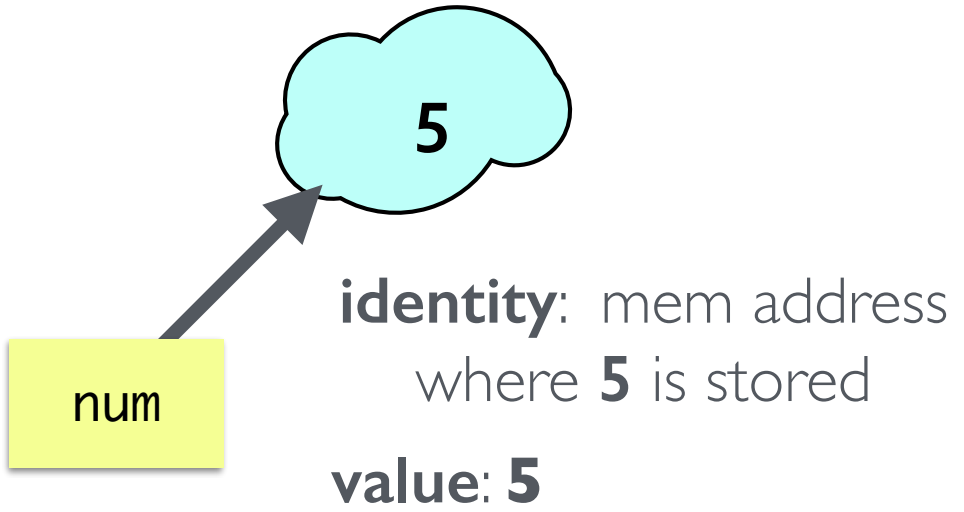
`[1, 3, 5, 6, 7, 11]`

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

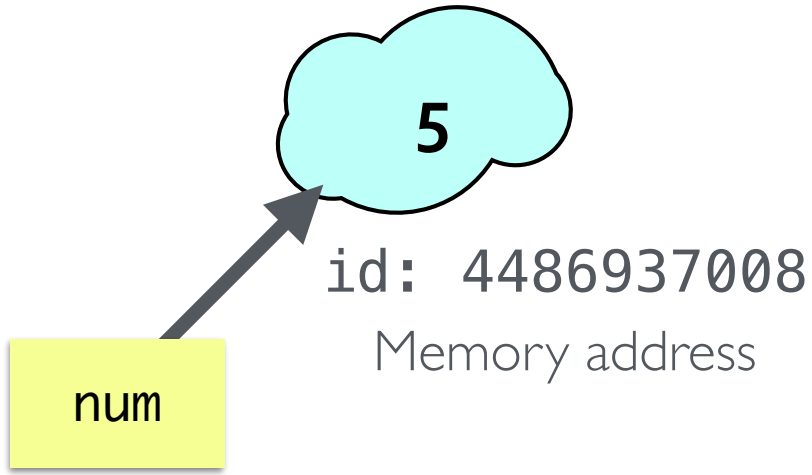
```
In [1]: num = 5

In [2]: id(num)
Out[2]: 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**

```
In [1]: num = 5

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

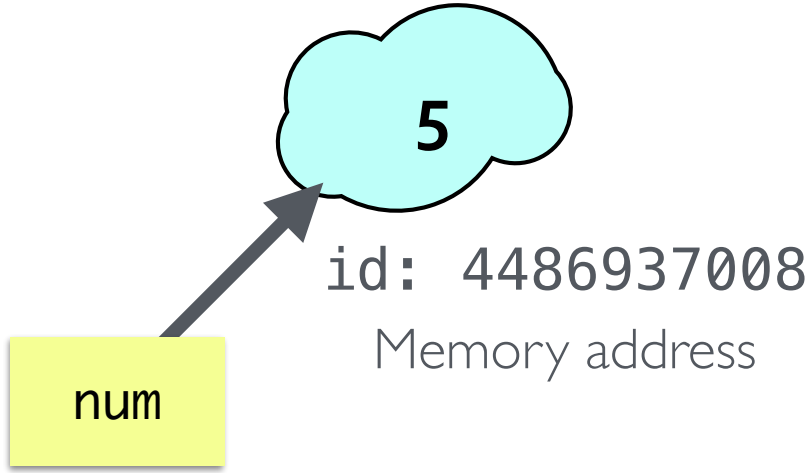


Variable names like `num` point to memory addresses of stored value

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

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

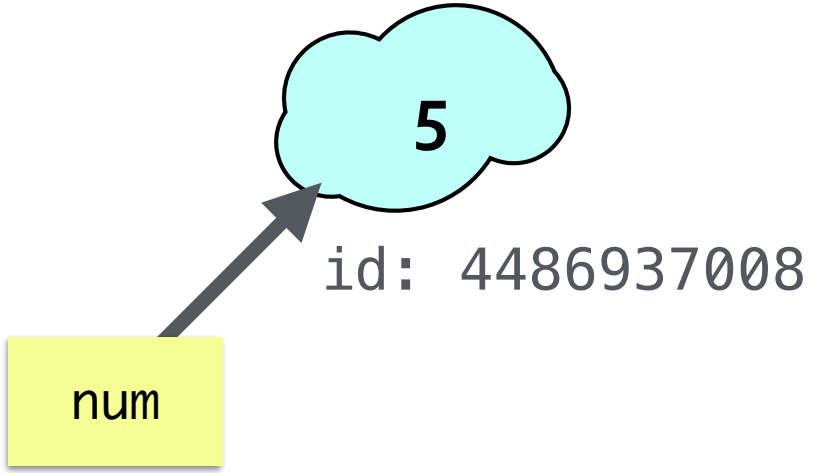
Lists are Mutable

- List values **can** be changed
- We just reviewed how we can mutate/change what's in a list using methods
- 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

Ints, Floats are Immutable

```
In [1]: num = 5

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

In [3]: num = num + 1

In [4]: id(num)
```

Has the identity of **num** changed?

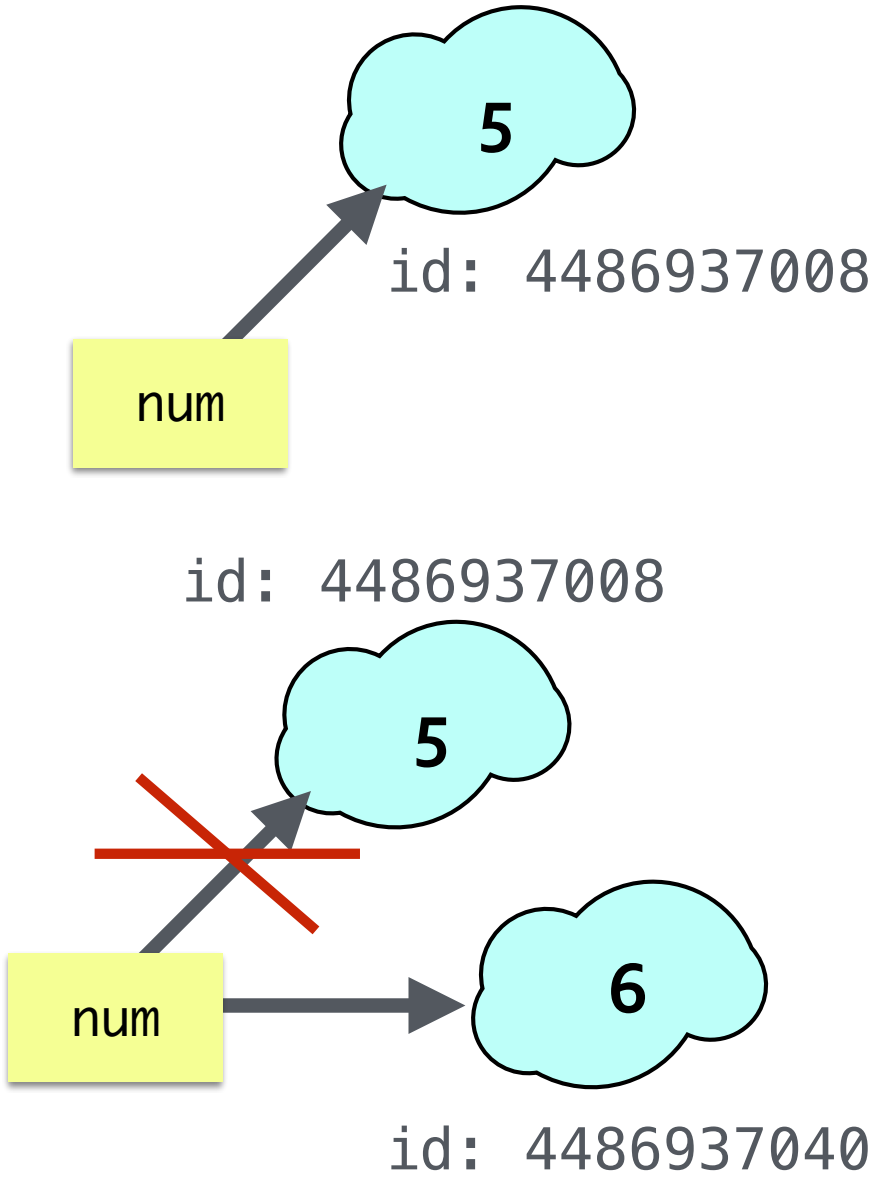


Attempts to change an immutable object creates a new object

Ints, Floats are Immutable

```
In [1]: num = 5

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

In [3]: num = num + 1

In [4]: id(num)
Out[4]: 4486937040
```

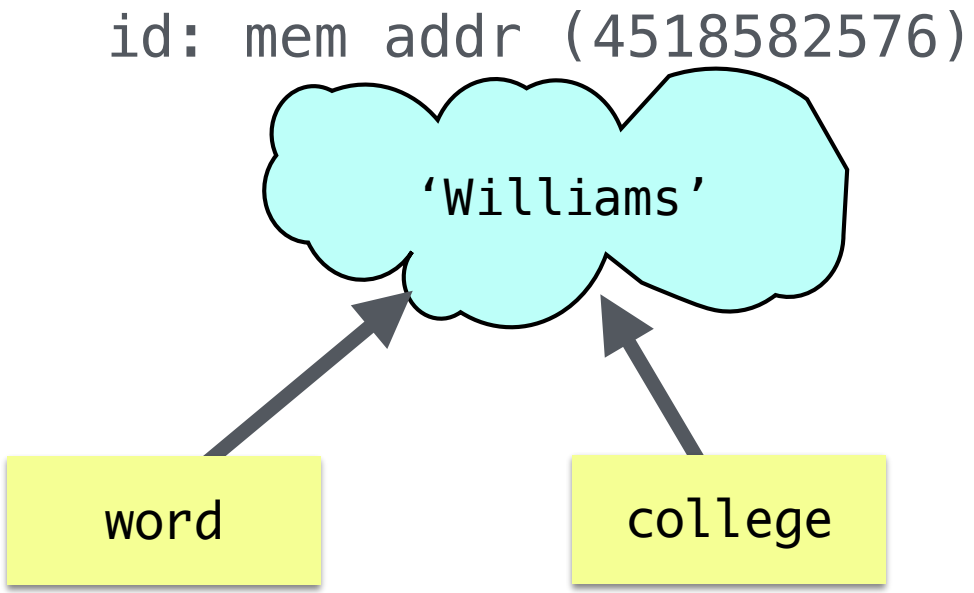
Identity of ints cannot be changed, **num** assumes a **new** identity



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
```



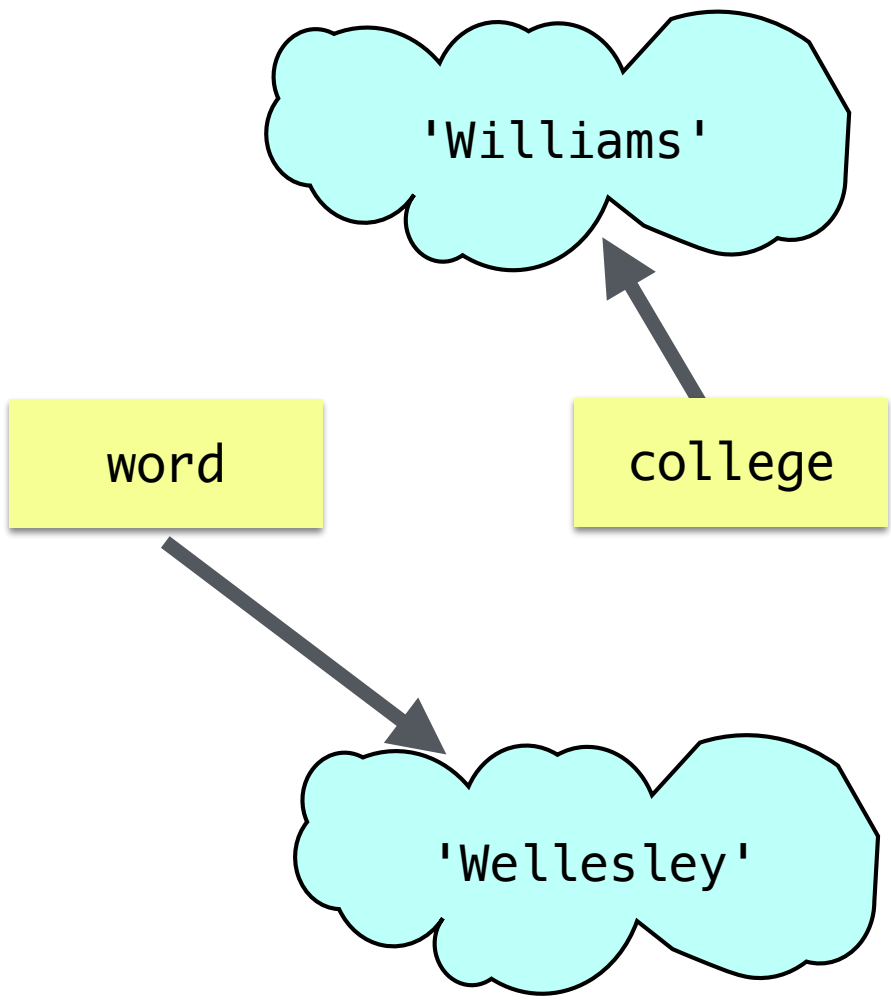
Variable names point to memory addresses of stored value

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
```

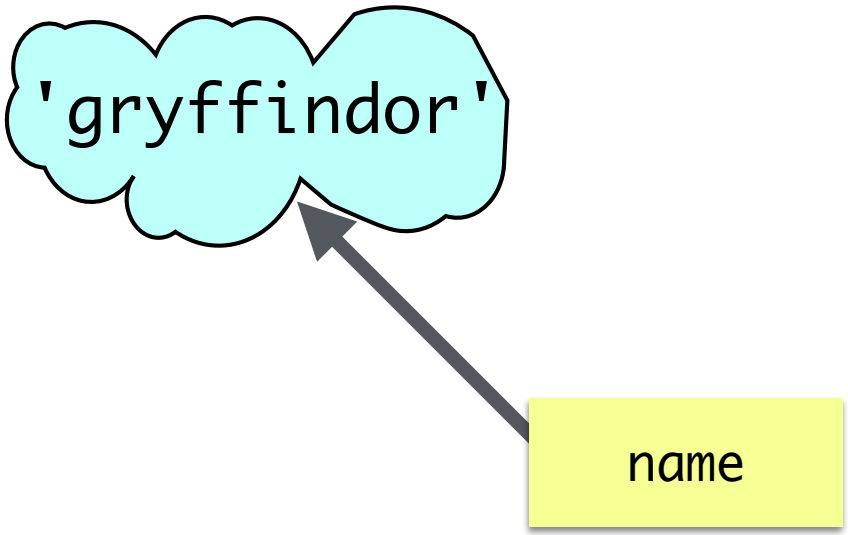


Attempts to change an immutable object creates a new object

String Methods/Operations Return New Strings

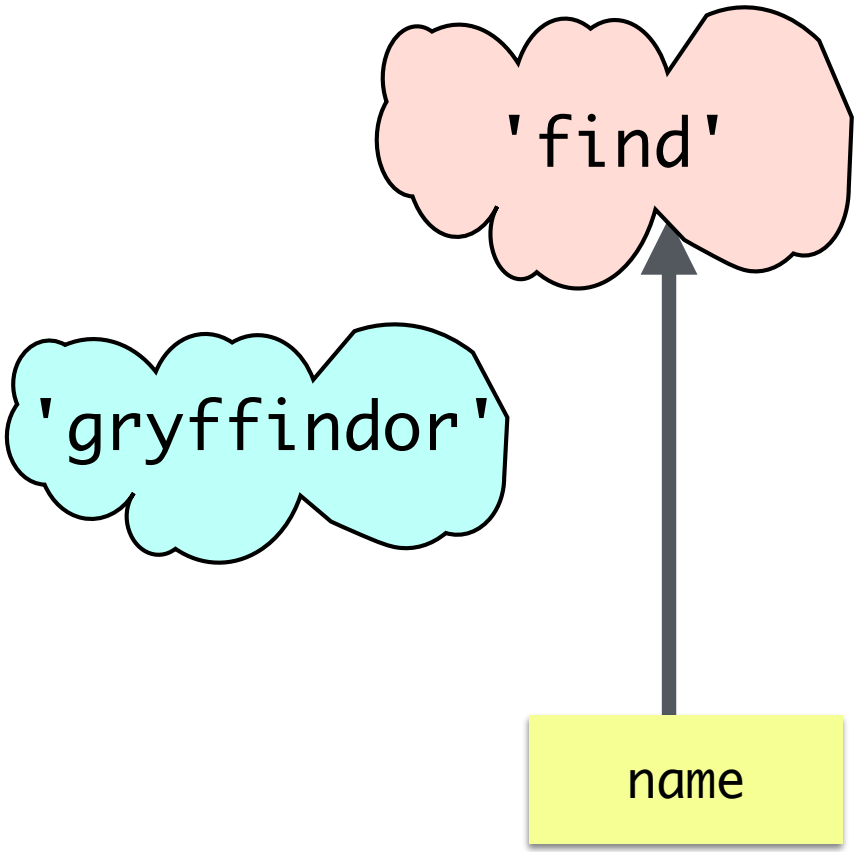
- String methods like `.lower(), .upper()` return a **new string**
- Sequence operations, like slicing `[:]`, return **new sequences**

```
In [1]: name = "gryffindor"

In [2]: id(name)
Out[2]: 4574657776
```



String Methods/Operations Return New Strings

- String methods like `.lower(), .upper()` return a **new string**
- Sequence operations, like slicing `[:]`, return **new sequences**




```
In [1]: name = "gryffindor"

In [2]: id(name)
Out[2]: 4574657776

In [3]: name = name[4:8]

In [4]: id(name)
Out[4]: 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 = 'Rohit and Jeannie'`
    - `myList = [1, 2, 3]`
  - concatenation (+) always creates a new sequence